

# 华清远见-成都中心-H5教学部



# 华清远见-成都中心-H5教学部

# 目录

JavaScript简介

为什么学习JavaScript

什么是JavaScript

引入JavaScript的三种方式

JavaScript输出数据到页面的方式

JavaScript输出数据到控制台的方式

# 目录

如何获取页面元素

如何获取和修改页面元素内容

如何改变元素的样式

引入JavaScript的三种方式

如何连接字符串

事件的概念及其三要素

# JavaScript简介

- 1992年Nombas的scriptease奠定了JavaScript思想理念；
- 受当时网速的限制，很多操作过程很漫长，如用户注册个账号，先提交到服务器去验证合法性，然后再返回给用户。Netscape发现了这个问题并开发了JavaScript语言，同时微软也开发了一个叫JScript语言，并且自己制定了一套规则。后来在ECMA（欧洲计算机制造商协会）、TC39（第39技术委员会）、W3C（World Wide Web Consortium）的共同努力下，制定了ECMAScript标准；
- JavaScript和Java是两种完全不同的语言，无论在概念还是设计上；当然Netscape和SUN公司当时是有合作关系的，在命名时也有借助JAVA这个明星提高自身知名度的意思。一句话，他们的关系就是雷锋和雷峰塔的关系。

# 为什么学习JavaScript

- JavaScript 能及时响应用户操作，被数以百万计的网页用来改进设计、验证表单、检测浏览器、捕捉用户动作等；
- HTML 定义了网页的内容，CSS 描述了网页的布局，JavaScript实现了网页的行为；
- JavaScript可以控制HTML的内容，可以增加、删除、修改、获取标签信息，直接在浏览器上修改标签属性或页面内容；
- JavaScript可以控制CSS样式，改变网页布局，比如改变元素背景色、文字大小、显示隐藏等；

# 什么是JavaScript

- JavaScript是一种具有面向对象能力的、解释型脚本语言；
- 基于对象和事件驱动、相对安全的客户端脚本语言；
- 运行在浏览器的JS解释器下；
- 在提供更好的用户体验上功不可没；
- 弱类型，定义变量时不必具有明确的数据类型；

## 引入JavaScript的三种方式

- 行级：给标签设置对应的属性，属性值是要执行的JS代码；
- 嵌入式：使用script标签，标签需要闭合，标签内容是要执行的JS代码；
- 引入式：使用script标签，标签需要闭合，设置属性src，src的值是js文件路径，如：./js/my.js；
- 嵌入或引入的数量是不受限制的；



# JavaScript输出数据到页面的方式

- 页面弹窗：

`alert('弹出的内容');`

弹出一个有确定按钮的信息框，多用于信息警告；

- 页面输出内容：

`document.write('输出到页面的内容');`

将内容输出到HTML文档中，如果文档加载完成后执行，则会覆盖掉所有原文档信息；

- 弹出选择框：

`confirm('你确定执行该操作?');`

点击“确定”按钮返回true；点击“取消”返回false；

# JavaScript输出数据到控制台的方式

- `console.log( '打印 "日志" 信息到控制台' );`
- `console.error( '打印 "错误" 信息到控制台' );`
- `console.warn( '打印 "警告" 信息到控制台' );`



- `console.table({name:'华清远见', age:14});`

| (index) | Value  |
|---------|--------|
| name    | "华清远见" |
| age     | 14     |

▼ Object 

age: 14  
name: "华清远见"

- 清空控制台消息：`clear();`

## 如何获取页面元素

- `document.querySelector( '选择器' )` :

这里的选择器和CSS样式定义的选择器是同一个概念;

如果该选择器对应多个元素, 则只返回第一个;

- `document.querySelectorAll('选择器')` :

获取选择器对应的全部元素;

返回值是类数组, 知道是类数组即可, 后面会深入讲解;

注意: 即便选择器只对应一个元素, 返回值也是类数组;

- 这里只需要知道这两种获取元素的方式即可, 后面会学习更多获取元素的方式;

## 如何获取和修改页面元素内容

- `document.querySelector('选择器').innerHTML` :  
获取指定选择的内容;
- `document.querySelector( '选择器' ).innerHTML = '新内容' ;`  
修改指定选择器对应的元素内容为新内容;
- 这里只需要知道这种操作方式即可，后面会学习更多获取和修改元素内容的方式；

# 如何改变元素的样式

- 语法是：`document.querySelector('选择器').style.属性 = '值';`
  - 属性是CSS样式中的属性，如`display`、`color`、`width`、`height`等；
  - 如果属性有横线(-)，如`background-color`、`font-size`、`border-radius`、`font-weight`，则把横线去掉，同时横线后面的第一个字母大写，  
如：`backgroundColor`、`fontSize`、`borderRadius`、`fontWeight`；
- 例子：
  - 隐藏元素：`document.querySelector('选择器').style.display = 'none';`
  - 改变字体颜色：`document.querySelector('选择器').style.color = '#FF0000';`
  - 改变背景颜色：`document.querySelector('选择器').style.backgroundColor = '#000000';`
  - 字体加粗：`document.querySelector('选择器').style.fontWeight = 'bolder'`

## 如何连接字符串

- 加号：+；

```
var name = '邓丽军';
```

```
var school = '成都中心';
```

```
var major = 'H5';
```

```
var s = name + '同学是' + school + major + '班的学生；';
```

# 事件的概念及其三要素

- 事件的概念

事件是指可以被JS监测到的网页行为；如：鼠标点击、鼠标移动、鼠标移入/离开、键盘按键、页面加载等；

- JavaScript事件的三要素：事件源、事件、事件处理

结合现实事件：小王，把灯打开一下；

事件源：操作对象，名词，对应：开关；

事件：做什么动作，动词，对应：摁一下；

事件处理：背后要做哪些工作，具体要干什么，这里就是我们要写代码具体实现的功能了，对应：接通火线，把灯点亮；

- 我们学习JS就是找到“事件源”并给他绑定“事件”，在事件发生时启动“事件处理”程序；

# 第16章 JavaScript词法结构

华清远见-成都中心-H5教学部





# 目录

基本概念

关于大小写

空格

注释

直接量

标识符与保留字

可选的分号

## 词法结构-基本概念

- 编程语言的词法结构是一套基础性规则，用来描述如何使用这门语言来编写程序；
- 词法结构是语法的基础，定义了如何定义变量、写注释、分割语句等基本规则；

# 词法结构-严格区分大小写

- JavaScript是严格区分大小写的
  - JavaScript区分大小写，包括关键字、变量、函数名、所有标识符；
  - querySelector的S是大写，你写成小写就会报错；
  - alert()全部是小写，你写一个大写字母就会提示你该函数不存在；
  - myname、myName、mynamE、MyName他们真不是一个东西；
- 大小写问题是新手常犯的一个错误，需要注意；

## 词法结构-空格

- JavaScript会忽略标识符前后的空格；
  - 空格是为了让代码有整齐一致的缩进，形成统一的编码风格，让代码更具可读性；
  - 你可以 `document . querySelector('选择器')` 这样；
  - 你还可以 `document. querySelector ('选择器')` 这样；
  - 但是你不可以 `document. query Selector('选择器')` 这样；
  - 所以，你要搞清楚JavaScript是忽略标识符前后的空格；
  - 在标识里面加空格，是把一个标识符分割成了两个或多个标识符；
- 一般加空格是为了代码排版，不要乱加空格；

# | 词法结构-注释

- JavaScript支持两种注释方式；

- 单行注释：//这里是注释内容；
- 多行或段落注释：

/\*

\*这里是段落注释

\*可以写多行

换行可以不要星号(\*)

\*加星号(\*)就是为了注释的可读性

\*段落注释是不能嵌套的

\*/

- 注释部分不会执行，合理的注释能显著提高代码的可读性；
- 可以通过浏览器源文件看到注释内容，所以什么该注释什么不该注释要注意；

## 词法结构-直接量

- JavaScript中直接使用的数据值叫做直接量；
  - 12306;
  - '我要学习JavaScript' ;
  - true;
  - null;
  - [1, 2, 3, 4, 5];

## 词法结构-标识符

- 标识符就是一个名字;
- JS使用标识符对变量和函数及其参数进行命名;
- JS标识符必须以字母、下划线(\_)、美元符号(\$)作为开始符;
- 后续可以是字母、数字、下划线或美元符号(\$);
- 数字不允许作为首字符出现，以便可以轻易区分开变量和数字，如：12345就是个数字，不用考虑是个变量的情况;
- 合法的标识符：myname、\_age、\$classname、abc、hqyj\_h5;
- 不合法的标识符：5myname;
- 也可以使用非英语来定义标识符：var  $\pi$  = 3.14 ,  $\alpha$  ;
- 标识符最好见名知意;

## 词法结构-保留字

- JavaScript默认的把一些标识符拿来作为自己的关键词，如：for、if、while，这些关键词称作保留字;
- 我们在写代码定义标识符的时候不能使用保留字;

|           |          |            |        |
|-----------|----------|------------|--------|
| break     | do       | instanceof | typeof |
| case      | else     | new        | var    |
| catch     | finally  | return     | void   |
| continue  | for      | switch     | while  |
| debugger* | function | this       | with   |
| default   | if       | throw      | delete |
| in        | try      |            |        |



## 词法结构-保留字

- 更多保留字：\* 标记的关键字是 ECMAScript5 中新添加的。

|          |           |            |           |              |
|----------|-----------|------------|-----------|--------------|
| abstract | arguments | boolean    | break     | byte         |
| case     | catch     | char       | class*    | const        |
| continue | debugger  | default    | delete    | do           |
| double   | else      | enum*      | eval      | export*      |
| extends* | false     | final      | finally   | float        |
| for      | function  | goto       | if        | implements   |
| import*  | in        | instanceof | int       | interface    |
| let      | long      | native     | new       | null         |
| package  | private   | protected  | public    | return       |
| short    | static    | super*     | switch    | synchronized |
| this     | throw     | throws     | transient | true         |
| try      | typeof    | var        | void      | volatile     |
| while    | with      | yield      |           |              |

## 词法结构-分号

- JavaScript使用分号(;)将语句分隔开；
- 分号并不是必须的，换行也可以表示一个语句结束，执行时会自动填充分号；
- JavaScript并不是在所有的换行处都填充分号：只有在缺少了分号无法正确解析时才会填充分号；
- 建议养成以分号结束语句的习惯，使用分号来明确标记语句的结束，即使在并不完全需要分号的时候也是如此；

# 华清远见-成都中心-H5教学部

# 目录

变量的基本概念及类型

JS中的变量类型及类型转换

不可变的原始值和可变的对象引用

变量的声明

变量的作用域及提升

作为属性的变量

# 基本概念及类型分类

- 一些基本概念
  - 在编程语言中，能够表示并操作的值(value)的类型，称为数据类型(type)；
  - 变量(variable)是一个值的符号名称，可以通过变量引用值；
  - 可以把一个值赋值给一个变量(variable)， 这样程序在任何需要使用该值的地方，就可以直接引用该变量；
- JavaScript的数据类型分类：
  - 原始类型：数字、字符串、布尔值、null(空)、undefined(未定义)；
  - 对象类型：原始类型之外的类型，如数组、对象、函数等；

## 变量类型-数字

- JS不分整数和浮点数，所有数字均用浮点数字表示；
- 直接出现在程序中的数字类型的直接量称为数字直接量；
- 数字直接量分为整型直接量和浮点型直接量；
- 整型直接量支持十六进制：0x作为前缀，其后跟0-9A-F；
- 浮点型由整数部分、小数点、小数部分组成,支持指数计数法，如:3.14e13；
- 数字直接量前面加负号(-)，得到它的负值；
- 需注意：负号(-)不是数字直接量的一部分，负号(-)是求反运算符；
- 数字支持的运算符有：+、-、\*、/、%(取余)；Math对象定义的方法可以实现更复杂的操作，如：Math.random()、Math.ceil()等，后面在讲对象时会详细讲解；
- Infinity表示正无穷大，-Infinity表示负无穷大；
- isNaN()判断一个值是不是数字，如：isNaN(5)，返回false；

## 变量类型-数字-存在的错误

- 实数有无数个;但是,JS只能用浮点数表示其中有限的实数,因为JS采用的是二进制表示法,如:1/2,1/4,1/8,1/256,我们常用的是十进制,如1/10,1/100,1/10000,那么如何精准的表示0.1这种非常简单的数字那?答案是并不能;
- 在JS中使用实数的时候,通常只是真实值的一个近似表示;
- 如何解决:可以先全部转成整数,运算完后再转回;
- 运行下面的代码试试看:

```
var a = 0.3 - 0.2;  
var b = 0.2 - 0.1;  
console.log(a);  
console.log(b);  
console.log(a==b);
```

## 变量类型-字符串

- JS通过字符串类型来表示文本;
- 字符串的索引从0开始，第一个字符的位置是0，第二个是1，以此类推；
- 长度为1的字符串代表一个字符，JS里面并没有“字符型”；
- 字符串直接量是指用单引号或双引号括起来的字符序列；
- 单引号括起来的字符串可以包含双引号；双引号括起来的字符串可以包含单引号；
- 一个字符串可以定义在多行上，建议在非常必要的时候才这么写：

```
'my name is farsight,\n  i am from chengdu,\n  i am a IT';
```



## 变量类型-字符串-转义字符

- 反斜线(\)后面加上一个字符，就不再表示字符的字面含义了，此时称为转义字符；
- \' 表示对单引号进行转义，这样就可以在单引号括起来的字符串中有单引号了；
- 常见的转义字符：
  - \t: 水平制表符；
  - \n: 换行符；
  - \r: 回车符；
  - \": 双引号，而不是字符串分界符；
  - \': 单引号，而不是字符串分界符；
  - \\: 反斜杠；

## 变量类型-字符串的一些操作

- 字符串的连接：+;
- 求字符串的长度：str.length ;
- 指定位置的字符：str[2]表示第三个字符;
- 字符串是固定不变的，对其操作时，会返回一个新的字符串，原始字符串不变；
- 字符串的更多操作在学习对象及正则表达式时讲；
- 参考代码：

```
var str = 'hello';  
console.log(str[2]);  
console.log(str.length);
```

## 变量类型-布尔值

- 布尔值指对或错、开或关、是或否;
- 布尔类型只有两个值：true 和 false，是保留字，小写；
- 布尔值通常用于控制语句中，如if、for、while等；
- JS的任意值都可以转换为布尔值，可直接使用Boolean()进行转换，下面六个值会转换为false：
  - undefined
  - null
  - 0
  - -0
  - NaN
  - ""
- 其他值则转换为true，包括空数组、对象等；

# 变量类型-null和undefined

- null

- 是保留字，常用来描述空值；
- `typeof null`: 返回的是字符串`object`，也就是说可以把`null`看成一个特殊的对象；
- 通常来讲我们把`null`看成他自有类型的唯一成员；

- undefined

- `undefined`表明变量没有初始化；
- 如果函数没有返回值，则返回`undefined`；
- `typeof undefined`: 返回的是字符串`undefined`；
- `==`认为`null`和`undefined`是相等的；`===`则返回`false`；

## 变量类型-对象

- JS对象是一种复合值，是属性或已命名值的集合，也就是键值对集合；
- 当值是函数的时候，我们称他为方法；
- 使用大括号定义对象：

```
var person = {  
    name:'yourname',  
    say:function(){  
        console.log('我要讲两句了');  
    }  
}
```

- 通过点(.)访问对象的属性：person.name；
- 通过点(.)访问对象的方法：person.say()；

## 变量类型-全局对象

- 全局对象是全局定义的符号，在程序中可以直接用;
- JS解释器启动时，他将创建一个新的全局对象；
- 常见全局对象：
  - 全局属性：如undefined、Infinity、NaN等；
  - 全局函数：如parseInt()、eval()、isNaN()、toFixed()等；
  - 构造函数：如Array()、String()、Date()等；
  - 全局对象：如Math；
- 在代码最顶级，可以用关键词this来引用全局对象，也可以用window自身属性来引用；  
如：this.alert(1); window.alert(1);
- 当我们声明一个全局变量时，这个全局变量就会作为全局对象的一个属性； var a = 8 ; window.alert(this.a);
- 我们在写程序时最好把全局对象当做保留字，避免出现意外错误；

## 变量类型-包装对象

- 存取字符串、数字或布尔值的属性时创建的临时对象称作包装对象;
- 如何理解临时对象：先看一段代码

```
var str = 'hqyj h5';  
str.len = 6;  
var l = str.len;
```

- 这段代码运行完成后l的值是undefined，why？

//创建一个原始类型：字符串 var str = 'hqyj h5';

//引用字符串属性时，调用new String(str)把str转换为对象：str.len = 6;

//一旦属性引用结束，新创建的对象会销毁，该对象属性也就不存在了；

//所以这里再引用时就是一个不存在的属性 var l = str.len;

//打印：undefined console.log(l);

看得出来，修改只是发生在临时对象身上，原始字符串并没有改变；

# 不可变的原始值和可变的对象引用

- 不可变的原始值

- 任何方法都无法更改一个原始类型的值:数字、字符串、布尔值、null、undefined;
- 对字符串类型的修改操作, 只会返回一个新的字符串, 原始字符串不会被修改;
- 原始值在做比较时, 只要值相等, 他们就是相等的;

- 可变的对象引用

- 对象类型的值是可以修改的, 看代码;

```
var obj = {name:'yourname', age:20};
```

```
obj.age = 30;
```

```
console.log(obj.age);
```



## 数据类型-对象的比较

- 包含相同属性及相同值的两个对象类型的值是不相等的：

- 我们通常将对象称为引用类型(reference type)；
- 依照术语叫法，对象值都是引用，对象的比较均是引用比较；
- 所以，当且仅当他们引用同一个对象时，才相等；

```
var a = [1,2,3];
```

```
var b = a;//变量b也引用这个数组
```

```
b[2] = 9;//通过变量b来修改引用的数组，变量a也会被修改
```

```
console.log(a === b);
```

```
console.log(a);//这个时候控制台会打印出来：[1,2,9]
```

- 将对象或者数组赋值给一个变量时，是对值的引用，本身并没有复制一份；
- 如果想复制一个副本出来，则需要显式的复制对象的每个属性或者数组的每个元素；

- 如果我们想比较两个对象或者数组的值是否相等，则需要比较他们的属性或元素；数组可以循环比较每一个元素；

# 数据类型-类型转换

- JS总是把值自动转换为它需要的数据类型;
- 转化为数字：
  - `-`(减)、`*`(乘)、`/`(除)，会把类型转换成数字;
  - `true`转化为1, `false`、空字符串转换为0;
  - 加号`+`，只要有一个是字符串，则全部作为字符串； 因为加号`(+)`同时还是字符串连接符号；
  - `parseInt()`、`parseFloat()`函数;
- 原始值到对象的转换：
  - 直接使用`String()`、`Number()`、`Boolean()`构造函数转换;

## 数据类型-变量声明

- JavaScript中使用一个变量之前需要先声明；使用没有声明的变量会报错;
- 使用var来声明一个变量：JS是弱类型语言，不需要设置类型
  - 一次声明一个变量：var name; var age;
  - 一次声明多个变量 var a, b, c;
  - 如果var声明的变量没有赋值，则初始值是undefined;
  - 声明变量的同时可以赋值（初始值）

```
var h = '华清远见';  
var q=1, y=2, j=3;
```
- 给一个没有声明的变量赋值是可以的，然而能不这么干就不要这么干；请养成使用var声明变量的好习惯；

# 数据类型-变量作用域及提升

- 变量作用域

- JS变量的作用域分全局变量和局部变量;
- JS中声明的全局变量是全局对象的属性;
- 函数体内(var声明)的变量称为局部变量, 函数的参数也是局部变量;
- 函数体内不使用var声明而直接赋值的变量当做全局变量;
- 函数体内局部变量的优先级高于全局变量;

- 变量提升

- 变量的提升是指会把变量的声明提升到前面, 但是不提升变量赋值;
- 解释器在执行js代码的时候, 会把所有的声明, 包括变量和函数的声明, 提到最前面;
- 变量的提升本质上是声明的提升;

## ■ 数据类型-作用域链

- JS里的全局代码或者函数，都有一个与之对应的作用域链；
- 作用域链可以理解为对象的列表，或叫对象的链表，他们是按优先级顺序排列的；
- 这些对象定义了一段代码或者函数的“作用域中”的变量；
- 在全局(函数外的)代码中，作用域链由一个全局对象组成；
- 在无嵌套的函数中，作用域链有两个对象：
  - 定义函数参数和局部变量的对象；
  - 定义全局变量的对象；如果是嵌套函数，则作用域链上至少三个对象；
- 当代码运行需要变量解析(就是查找需要的变量的值)的时候，就在作用域链(有顺序的对象或者链表)里面寻找对应的变量，一旦找到就使用该变量并结束寻找；如果作用域链里面找不到对应的变量，则抛出错误；

## 数据类型-作为属性的变量

- 使用var声明变量时，作为全局对象的一个属性，无法通过delete运算符删除；
- 不使用var，直接给一个未声明的变量赋值，该变量是可以delete的；
- 不使用var，使用关键词this给一个变量赋值，可以delete；

```
this.a=1;
```

```
var r = delete a;
```

```
console.log(r);//true
```

```
console.log(a);//报错
```

```
var b = 90;
```

```
console.log(delete b);//false
```

# 第18章 表达式和运算符

华清远见-成都中心-H5教学部



# 目录

原始表达式

数组初始化表达式

对象初始化表达式

函数定义表达式

属性访问表达式



# 目录

调用表达式

算术表达式

关系表达式

逻辑表达式

赋值表达式

# 表达式和运算符-原始表达式

- 原始表达式不可再分割，是最小单位的表达式；简单的概念性东西，知道即可；
- 原始表达式包含直接量、关键字(保留字)和变量名；
- 常见原始表达式举例：

//直接量

1;

1.02;

'hello world!';

//保留字

true;

false;

this;

//变量名

name; hqyj; age;

## 表达式和运算符-数组初始化表达式

- 可以简单理解为：数组直接量；
- 数组初始化表达式：由中括号([])和其内用逗号(英文状态 ,)分隔开的列表构成；
- 初始化的结果是创建一个新的数组；数组的元素是逗号分隔开的表达式的值；
- “数组初始化表达式”中的“元素”也可以是“数组初始化表达式”，即多维数组；
- 常见数组初始化表达式举例：

- [];
- [1,2,3,4,'a','b','c'];
- [['a','b','c'],['d','e','f'],['g','h']];

//中间省略的元素会怎样？最后省略的元素又会怎样？

- [1,,,6,,]

## 表达式和运算符-对象初始化表达式

- 可以简单理解为：对象直接量；
- 对象初始化表达式：由花括号({})和其内用逗号(英文状态 ,)分隔开的列表构成；
- 初始化的结果是创建一个新的对象； 对象的属性是逗号分隔开的表达式的值； 属性包括属性名和属性值，属性名和属性值之间用冒号隔开；
- “对象初始化表达式” 中的 “元素” 也可以是 “对象初始化表达式” ，可以任意层级嵌套；
- 对象初始化表达式举例：

```
{name:'yourname', age:22};  
  
{  
    p1:{name:'华子', age:22},  
    p2:{name:'华耀', age:25}  
};
```

## 表达式和运算符-函数定义表达式

- 可以简单理解为：函数直接量；
- 表达式的值是这个新定义的函数；
- 函数表达式举例：

```
var fn = function(a, b) {  
    return a+b;  
}
```

## 表达式和运算符-属性访问表达式

- 有两种访问方式：点(.)、中括号([])；
- 点的访问方式更方便，中括号的访问方式适用所有情况；
- 参考例子：

```
var a = {name:'yourname', age:22};
```

```
var b = {
```

```
  p1:{name:'华子', age:22},
```

```
  p2:{name:'华耀', age:25}
```

```
};
```

```
var c = [1,2,3,4,'a','b','c'];
```

```
var d = [['a','b','c'],['d','e','f'],['g','h']];
```

```
console.log(a.age);
```

```
console.log(b.p2['name']);
```

```
console.log(c[4]);
```

```
console.log(d[2][1]);
```

## 表达式和运算符-调用表达式

- 如果调用的函数或方法有返回值，则表达式的值就是该返回值；
- 如果调用的函数或方法没有返回值，则表达式的值是undefined；
- 简单理解就是函数或方法名称后面跟上小括号代表执行；
- 例子：

`fn(1, 2);`

`Math.max(1,2,3);`

`a.sort();`

## 表达式和运算符-对象创建表达式

- 由函数调用表达式延伸而来，前面加个new即可；
- 如果不需要传递参数，函数后面的小括号可以省略；
- 如果调用的函数或方法没有返回值，则表达式的值是undefined；
- 简单理解就是函数或方法名称后面跟上小括号代表执行；
- 例子：

`new Array(1,2,3);`

`new String('hello world!');`



## 表达式和运算符-算术表达式

- 算术运算符：加减乘除取余，+ - \* / %
- +：数字相加 或 字符串连接；
  - 如果其中一个操作数是对象，则JavaScript会自动把他转成原始类型的值；
  - 如果其中一个操作数是字符串的话，则另一个也会转成字符串，此时是字符串连接；
  - 如果两个操作数都是数字，则进行加法运算；
- 例子：

```
console.log('1' + 5);           //15
console.log(12 + NaN);          //NaN
console.log(true + true);       //2
console.log(201 + null);        //201
console.log(203 + undefined);   //NaN
console.log(3 + 5 + '猜猜看');  //8猜猜看
```

# 表达式和运算符-一元算术运算符

- +：可以拿来很方便的把操作数转换为数字；
- -：把操作数转换成数字，改变运算结果的符号；
- ++、--：自增自减运算符，常用于循环的计数器控制；
  - 放在操作数前面表示先运算后执行；
  - 放在操作数后面表示先执行后运算；

- 例子：

```
var a = 2;
var b = 3;
var c = (++a) + (b++) + b + (a++) - a;
console.log(c);    //9
var a = 2;
var b = 3;
var c = (++a) + (b++) + b + (a++) - a;
console.log(c);
```

# 表达式和运算符-位运算符

- 位运算符对数字的二进制数据进行更低层级的按位运算；
  - 位运算要求操作数是整数；
  - 位运算会将NaN、Infinity、-Infinity转换为0；
  - 按位与：&，对操作数的二进制表示逐步执行AND操作；
  - 按位或：|，对操作数的二进制表示逐步执行OR操作；
  - 按位异或：^，对操作数的二进制表示逐步执行XOR操作，一个为1另一个不为1才返回1；
  - 按位取反：~，对操作数的二进制表示所有位取反；
  - 左移：<<，对操作数的二进制表示进行左移，移动位数由第二个操作数指定；新的一位用0补充，并舍弃第32位；左移1位相当于\*2，左移2位相当于\*4，以此类推；
  - 带符号右移：>>，右边溢出的位忽略；如果是正数，左边最高位补0；如果是负数，左边最高位补1；右移1位相当于/2，右移2位相当于/4，不要余数，以此类推；
  - 无符号右移：>>>，和带符号右移一样，只是左边的最高位总是补0；

## 表达式和运算符-关系表达式

- 总是返回一个布尔值true或false；
- ==：相等运算符，检查两个值是否相等，不考虑类型；
- ===：恒等运算符，检查两个值是否相等，同时考虑类型；
- 比较运算符：>、>=、<、<=；
- in运算符：检查右侧对象里面是否拥有左侧属性名，如果有返回true；

```
var a = {x:1, y:2, z:"};
```

```
console.log('x' in a);           //true
```

```
console.log('z1' in a);          //false
```

```
console.log('toString' in a);    //true
```

```
console.log('isPrototypeOf' in a); //true
```

## 表达式和运算符-关系表达式

- instanceof:检查左侧的对象是否是右侧类的实例，如果是返回true
- 如果一个对象是一个“父类”的子类的实例，则返回true；
- 所有的对象都是Object的子类，后面原型链会讲到；
- 例子：

```
var d = new Date();  
console.log(d instanceof Date); //true  
console.log(d instanceof Array); //false  
console.log(d instanceof String); //false  
console.log(d instanceof Object); //true
```

## 表达式和运算符-逻辑表达式

- 逻辑&&：都为true时才返回true；一旦遇到false，后面的代码不会执行；
- 逻辑||：有一个为true时返回true；一旦遇到一个true，后面的代码不会再执行；
- 逻辑!：首先将操作数转换为布尔值，然后对布尔值取反；
- 如何理解后面的代码不再执行：

```
var c = (a = 3) || (b = 4);
```

```
console.log(a);    //3
```

```
console.log(c);    //3
```

```
console.log(b);    //代码报错，不再往下执行
```

## 表达式和运算符-赋值表达式

- 赋值： $=$
- 带运算的赋值： $+=$ 、 $-=$ 、 $*=$ 、 $/=$ 、 $\%=$ 、....

# 表达式和运算符-表达式计算

- `eval()`
  - `eval()` 可以解释运行由JS源代码组成的字符串，并产生一个值；
  - 如果你使用了`eval()`，你要确定你确实需要使用它；



## 表达式和运算符-更多运算符

- 三元运算符：条件运算符(?:)，结构：条件?条件为true执行:条件为false执行；
- 获取操作数类型:typeof，放在操作数前面(也可以使用小括号)，得到操作数类型(字符串)；参考代码：

| X           | typeof X    |
|-------------|-------------|
| undefined   | "undefined" |
| null        | "object"    |
| true  false | "boolean"   |
| 数字  NaN     | "number"    |
| 字符串         | "string"    |
| 函数          | "function"  |
| 内置对象        | "object"    |

## 表达式和运算符-更多运算符

- delete：用来删除对象属性或数组元素；
- void：忽略计算结果并返回undefined；
- 逗号运算符：用于分割语句，返回的是最后一个语句的值；

//使用逗号运算符声明变量

```
var a = 1, b=2, c=3;
```

```
var d = function () {
```

```
    return (
```

```
        function () { return 1; }(),
```

```
        function () { return 2; }()
```

```
    );
```

```
}();
```

```
console.log(d); //2
```

# 华清远见-成都中心-H5教学部



# 目录

语句基本概念

复合语句

空语句

声明语句

条件语句

循环语句与跳转语句

严格模式

# 语句-基本概念

- 语句(statement)就是JS的命令
  - 语句以分号(;)结束，也可以换行作为新的语句开始；
  - 简单理解就是告诉JS解释器要干什么；
  - 再简单理解下，所谓JS程序就是一系列语句的集合；
  - JS解释器在执行程序的时候，就是按编写顺序来执行语句，当然JS解释器会把声明提升到前面；
  - 前面讲到的各种表达式就是表达式语句；

## 语句-复合语句

- 复合语句就是用大括号将多条语句括起来；
- 语句块的结尾不需要分号；
- 整齐缩进的代码看上去更安逸，更容易理解；

```
{  
    var now = new Date();  
    var month = now.getMonth() + 1;  
    var day = now.getDate();  
    console.log('今天是'+month+'月'+day+'号');  
}
```

- 当需要把多条语句当做一条语句来执行时，用复合语句，如循环：

```
var age = 1;  
while (age < 100) {  
    document.write('我今年' + age + '岁了； ');  
    age++;  
}
```

## 语句-空语句

- 空语句就是只有语句的结束符分号(;), 没有其他代码;
- ; 这就是一个空语句, 只有分号。
- 例子:

```
var h = new Array(5);  
console.log(h.length);           //5  
for (var i = 0; i < h.length; h[i++]=i);  
console.log(h);                  //[1, 2, 3, 4, 5]
```

# 语句-声明语句

- 用于声明变量的var
  - var之后跟随的是要声明的变量列表;
  - 可以指定或不指定初始值, 不指定初始值则默认值是undefined;
  - 在函数体内用var声明的变量是局部变量;
  - 在全局代码中用var声明的变量是全局变量;
  - var声明的变量无法用delete删除;



# | 语句-声明语句

- 用于定义函数的function

- 函数声明的语法格式如下：

//functionname是函数名称，如：addnum

//arg1, arg2, arg3... 称为函数的形参，可有可无

//函数体的大括号是必须的，里面可以放任意多的语句

```
function functionname(arg1, arg2, arg3...){  
    statements  
}
```

- 在定义函数时，并不执行函数体内的语句；
  - 函数名后跟小括号，代表执行函数，也就是会执行函数体(大括号)里面的语句；
  - 和变量提升类似，function声明的函数(名及其函数体)会提升到作用域最前面；
  - 默认返回值undefined，如果需要指定返回值使用return；

# | 语句-条件语句

- 条件语句是通过判断表达式的值来确定是执行还是跳过某些语句(分支)；
  - if(如果):
    - 第一种形式: `if(expression){ statements; }`
    - 第二种形式: `if(expression){ statements 1; } else{ statements 2; }`
    - 括住表达式的圆括号是必须的；
    - 括住语句主体的大括号在只有一个语句的时候，是可以省略的；  
但是往往我们总是给语句主体加上大括号，即便只有一个语句；  
这样可以很好的避免语句歧义及阅读困难问题；
  - else if: else和if之间有空格
    - 当有多个分支时我们使用else if实现；
    - else if并不是真正的JS语句，他是多个if/else连在一起的惯用写法；

# | 语句-条件语句

- switch:适用于多个分支依赖于同一个表达式；
  - switch语句的语法结构：switch(expression){ statements;}；
  - 代码块中可以使用多个由case关键字标记的代码片段； case后面跟的是一个表达式和一个冒号；

```
var expression = 'status2';
switch(expression){
    case 'status1':
        代码片段1;
        break;
    case 'status2':
        代码片段2;
        break;
    default:
        默认代码片段;
        break;
}
```

## 语句-条件语句

- 如果其中一个case子句中的表达式和条件表达式相同(===)， 则从对应的代码块开始执行；
- 不要简单理解为执行对应的代码块， 这就是为什么代码块最后要加break跳出switch的原因： case只标记开始位置， 不标记结束位置；
- 如果找不到和条件表达式相同的case， 则执行default下面的代码块；

```
var a = 3;
switch(a){
    case 2:
        console.log('华清远见');
        break;
    case 3:
        console.log('成都');
    case 4:
        console.log('中心');
        break;
}
```

# | 语句-循环语句

- while循环语句

- while循环语句的语法结构：while(expression){ statement};
- 不断的执行statement主体语句，直到expression为false;

- do...while循环语句

- 语法结构：do{ statement}while(expression);
- 至少执行一次；用分号结尾；
- 不断的执行statement主体语句，直到expression为false;

- 小练习：

假设某人有100,000现金。 每经过一次路口需要进行一次交费；  
交费规则为： 当他现金大于50,000时每次需要交5%；  
如果现金小于等于50,000时每次交5,000；  
请写一程序计算此人可以经过多少次这个路口。

# 语句-循环语句

- for循环语句

- for循环的语法结构:

for(初始化语句; 循环判断条件; 计数器变量更新){ statement;}

不断的执行statement主体语句，直到判断条件不成立;

- 循环小案例：

- 循环一个数组;
  - 写个乘法口诀表;
  - 百钱百鸡:

公鸡5文钱一只， 母鸡3文钱一只， 小鸡1文钱三只;

用100文钱买100只鸡;

问，公鸡，母鸡，小鸡各几只？

# | 语句-循环语句

- for..in循环语句

- for...in循环的语法结构：  
for(variable in object){ statement;}
- for...in循环出来object里所有属性的属性名或者元素的下标；

- 实例：

- 循环对象：  

```
var personobj = {  
    name: '你叫什么？',  
    sex: '男',  
    weight: '160'  
};  
for (var x in personobj) {  
    document.write('属性:' + x + ', 值:' + personobj[x] + '<br />');  
}
```

## 语句-循环语句

- 循环数组：

```
var arr = [1,2,3,4,5];  
for(var i in arr){  
    console.log(i);  
    console.log(arr[i]);  
}
```

- 分析下面的代码执行结果：

```
var personobj = {  
    name: '你叫什么？',  
    sex: '男',  
    weight: '160'  
};  
var arr = new Array();  
var i = 0;  
for(arr[i++] in personobj);  
console.log(arr);
```



## 语句-跳转语句-标签语句

- 跳转语句可以让JavaScript代码的执行从一个位置跳转到另一个位置；
- 标签语句
  - 标签由标识符加冒号组成，如：gohere：
  - 标签标识符+冒号(:)+语句，就组成了标签语句
  - break 和 continue可以使用语句标签
- break用于结束一个循环；
- continue用于结束当次循环，并继续执行下一次循环；

## 语句-跳转语句-break和continue

- 体会break后面加和不加语句标签的区别：

```
var a = 0;
forouter_b:
for (var i = 0; i < 10; i++) {
    for (var j = 0; j < 10; j++) {
        if(j == 5){
            break forouter_b;
        }
        a++;
    }
}
console.log(a);
```

- 体会continue后面加和不加语句标签的区别：

```
var a = 0;
forouter_c:
for (var i = 0; i < 10; i++) {
    for (var j = 0; j < 10; j++) {
        if(j == 5){
            continue forouter_c;
        }
        a++;
    }
}
console.log(a);
```

## 语句-跳转语句-return

- return指定函数调用后的返回值；
  - return只能在函数体内出现；
  - return后面的代码不会再执行；
  - 在调试函数时可以使用return指定断点位置；

## 语句-跳转语句-throw

- throw用于抛出异常：
  - 立即停止正在执行的程序，跳转至就近的逻辑异常处理程序；

```
function factorial (n) {  
    if(isNaN(n)) throw new Error('请输入数字，HOHO');  
    if(n == 1) return 1;  
    return n*factorial(n-1);  
}
```

```
var n = factorial(3);  
console.log(n);
```

```
var n = factorial('a05');  
console.log(n);
```

## 语句-跳转语句-try-catch-finally

- try-catch-finally是JavaScript的异常处理机制：

- 语法结构：

```
try{  
    //正常业务代码  
}  
catch(e){  
    //上面的业务代码报错了会执行这里  
    //console.log(e);  
}  
finally{  
    //这里总是会执行  
}
```

- 异常对象e里面有两个属性name和message，分别代表错误类型和错误描述信息；

## 语句-其他语句-with

- 用于临时扩展作用域，with代码块执行完成后恢复原始状态；
- 不推荐使用with，有代码不好优化，运行慢等问题；
- 严格模式下是不能使用with的；

## 语句-其他语句-debugger

- debugger语句会产生一个断点，用于调试程序，并没有实际功能；
- 使用debugger的前提是你手动打开了“开发者工具”；
- debugger会产生一个断点，代码会停止在这里不再往下执行；

## 语句-其他语句-use strict

- use strict是将后续的代码解析为严格代码；

- 严格模式下禁止使用with；
- 严格模式下所有的变量都要先声明；
- 严格模式下调用函数的this值是undefined；

```
"use strict";
```

```
function fn(){
```

```
    console.log(this);
```

```
}
```

```
fn();
```

- 严格模式和普通模式有很多细微差别，记住很困难，基本原则是：写代码要严谨；



# 华清远见-成都中心-H5教学部



# 目录

对象直接量

new创建对象

原型

属性操作

## 对象-基本概念

- 除了字符串、数字、null、undefined、true、false，JS里面的其他值都是对象；
- 字符串、数字、布尔值的行为和不可变对象非常类似；

## 对象-创建对象-对象直接量

- 由若干名/值对组成的映射表；
  - 名和值中间用冒号分开；
  - 多个名/值对之间用逗号(,)分号；
- 整个映射表用大括号括起来；

- 定义一个对象直接量：

```
var person = {  
    name: {  
        firstname: '鲁',  
        lastname: '班'  
    },  
    work: 'carpenter',  
    age: 28  
};
```

## 对象-创建对象-new创建对象

- new后面跟一个函数表示创建对象；这里的函数是构造函数(constructor)；
- 用new创建几个内置对象：

```
var obj1 = new Object();
```

```
console.log(obj1);
```

```
var obj2 = new Date();
```

```
console.log(obj2);
```

```
var obj3 = new Array();
```

```
console.log(obj3);
```

# 对象-创建对象-原型初识

- 一切皆对象，他们都来自null；
  - 每一个对象(除了null)都和另一个对象相关联，这个所谓的另一个对象就是原型；
  - 不难理解，原型也是对象；
  - 所有的内置构造函数都有一个继承自Object.prototype的原型；
  - Object.prototype是没有原型的对象；
  - 每一个对象都从原型继承属性，直到null结束；
  - 原型链概念：

`var arr1 = new Array(1,2,3);`

`arr1->Array.prototype->Object.prototype->null;` 形成链，到null结束；

`var d = new Date();`

`d->Date.prototype->Object.prototype->null;` 形成链，到null结束；

这种属性继承关系就形成了原型链；

# 对象-创建对象-原型初识

- 构造函数、原型和实例的关系

- 构造函数都有一个属性prototype，这个属性是一个对象，是Object的实例；
- 原型对象prototype里有一个constructor属性，该属性指向原型对象所属的构造函数；
- 实例对象都有一个\_\_proto\_\_属性，该属性指向构造函数的原型对象；

- prototype与\_\_proto\_\_的关系

- prototype是构造函数的属性；
- \_\_proto\_\_是实例对象的属性；
- 两者都指向同一个对象；

## 对象-创建对象-Object.create()

- Object.create()是一个静态函数；
  - 用于创建一个新对象，第一个参数是这个对象的原型；

- 实例：

```
var b = Object.create(new Array(1,4,5,6,2,3,7));  
console.log(b.length);           //7
```

```
var nu = Object.create(new String('null'));  
console.log(nu[2]);              //1
```

```
var nu = Object.create(null);  
console.log(nu[2]);              //undefined
```



## 创建对象-属性的查询、设置和删除

- 使用(.)和中括号([])访问和设置对象的属性；

- delete运算符可以删除对象的属性；

```
delete person1.name;
```

```
delete person1.age;
```

- 只能删除自有属性，不能删除继承属性；
- delete删除成功或删除不存在的属性或没有副作用时，返回true；

- 实例：

```
var x = 1;
```

```
delete this.x; //false
```

```
function my() {}
```

```
delete this.my; //false
```

```
this.a = 100; //没有var
```

```
delete a; //true
```

## 对象-检测属性

- 判断某个属性是否存在于某个对象中；
  - in: 检查一个属性是否属于某个对象，包括继承来的属性；
  - hasOwnProperty(): 检查一个属性是否属于某个对象自有属性，不包括继承来的属性；
  - propertyIsEnumerable(): 检查一个属性是否属于某个对象自有属性，且该属性可枚举，不包括继承来的属性；

- 实例：

```
var person = {name:'yourname', age:10};  
console.log('sex' in person);           //false  
console.log('toString' in person);      //true  
console.log(person.hasOwnProperty('name')); //true  
console.log(person.hasOwnProperty('toString')); //false  
console.log(person.propertyIsEnumerable('name')); //true  
console.log(person.propertyIsEnumerable('toString')); //false
```

## 对象-枚举属性

- for/in循环遍历对象中所有可枚举属性，把属性名称赋值给循环变量；
  - 对象继承的内置方法不可枚举；

- 实例：

```
var person = {name:'yourname', age:10};  
console.log(person.propertyIsEnumerable('toString'));    //false  
for (v in person) {  
    console.log(v);          //打印name和age，不打印toString  
}
```

# | 对象-属性getter和setter

- 存取器属性

- 属性值可以由一个或两个方法替代，这两个方法就是getter和setter；
- 由getter和setter定义的属性，称为“存取器属性”；
- 一般的只有一个值的属性称为“数据属性”；
- 查询存取器属性的值，用getter；拥有getter则该属性可读；
- 设置存取器属性的值，用setter；拥有setter则该属性可写；

- 实例：只读属性

```
var getnum = {  
    get octet(){return Math.floor(Math.random()*256);},  
    get uint16(){return Math.floor(Math.random()*65536);},  
    get int16(){return Math.floor(Math.random()*65536) - 32768;}  
};  
console.log(getnum.octet);  
console.log(getnum.uint16);  
console.log(getnum.int16);
```

## 对象-属性getter和setter

- 定义一个简单的有存取器属性的对象：

```
var myobj = {  
    myname:'yourname',  
    birthday:'1987-05-17',  
    get myage(){  
        return (new Date().getFullYear()) - new Date(this.birthday).getFullYear();  
    },  
    set myage(value){  
        this.birthday = value;  
    }  
};  
myobj.myage = '1998-08-22';  
console.log(myobj.myage);
```

# | 对象-序列化对象

- 序列化对象是指将对象的状态转成字符串，也可以将字符串还原为对象；

- 转成字符串：JSON.stringify();
- 还原为对象：JSON.parse();

- 实例：

```
var hqyj = {  
    name:'华清远见',  
    add:'科华北路99',  
    tel:['0900', 8304, '0910']  
};  
var str = JSON.stringify(hqyj);  
console.log(str);  
console.log(typeof str);  
var obj1 = JSON.parse(str);  
console.log(obj1);
```

## 对象-对象方法

- toString：返回调用该方法的对象的值的字符串；

- 实例：

//对象调用该方法返回 [object Object]

```
var o = {h:1}.toString();
```

```
console.log(o);
```

//数组调用该方法返回数组元素列表

```
var a = [1,2,3];
```

```
console.log(a.toString());
```

//函数调用该方法返回的是函数本身的代码

```
function hq (argument) {
```

```
    函数主体代码;
```

```
}
```

```
console.log(hq.toString());
```

## 对象-对象方法

- `valueOf()`方法和`toString()`方法类似，往往在需要讲对象转换为原始值时用到；

- 实例：

```
console.log(new Date().valueOf());    //返回毫秒数
console.log(new Date().toString());    //返回日期格式的字符串
var a = [1,2,3];
console.log(a.valueOf());              // [1, 2, 3]
```



# 第21章 数组

华清远见-成都中心-H5教学部



# 目录

数组的创建

元素的读写及添加删除

数组的遍历

多维数组

数组操作方法

# 数组-创建数组

- 数组是对象的特殊表达形式：
  - 数组的元素可以是数字、值、表达式、对象、数组等；
  - 定义数组的几种方式：
    - 中括号的方式，简单方便：

```
var arr1 = [1,2,3,45,8,6];  
console.log(arr1);
```
    - `new Array()` 创建数组对象：
      - 空数组  

```
var arr2 = new Array();
```
      - 指定长度的数组  

```
var arr3 = new Array(10);
```
      - 指定元素  

```
var arr4 = new Array(12, 34, 5,6,7,8);
```
      - 数组元素支持表达式  

```
var h = 10; var arr5 = new Array(h+1, h+5, h+9);
```
      - 元素可以是其他类型  

```
var arr6 = new Array({x:1, y:2}, h+5, h+9);
```

## 数组-元素的读和写

- 用中括号对数组元素进行读写操作：

```
var h = 10;
```

```
var arr6 = new Array({x:1, y:2}, h+5, h+9);
```

```
console.log(arr6[0]);
```

```
arr6[3] = 30;
```

```
console.log(arr6);
```

## 数组-稀疏数组

- 数组元素不连续的数组我们称为稀疏数组：

```
var arr = new Array(5);
```

```
arr[10] = 6;
```

```
arr[100] = 56;
```

```
console.log(arr);
```

## 数组-数组长度

- arr.length得到数组的长度：

```
var arr = ['h', 'q', 'y', "", "", 'j', "", "];  
console.log(arr.length);           //8  
arr.length = 0;  
console.log(arr);                  //[] ， 打印空数组
```

# 数组-元素的添加和删除

- 元素的添加：

```
var arr = ['h', 'q', 'y', 'j'];  
arr[4] = 'five';           //指定索引，添加元素  
arr.push('one');           //在末尾添加元素  
arr.push('two_1', 'two_2');  
arr.unshift('first');      //在开头添加元素  
arr.unshift('first_1', 'first_2'); //两个元素一次加入
```

- 元素的删除：

```
var arr = ['h', 'q', 'y', 'j'];  
delete arr[2];             //删除一个指定元素，注意和下面的区别  
arr.pop();                 //删除最后一个元素  
arr.shift();               //删除第一个元素  
delete arr[0];             //删除一个指定元素
```

## 数组-遍历

- for循环对数组元素进行遍历：

```
var arr = ['h', 'q', 'y', 'j'];  
for (var i = 0; i < arr.length; i++) {  
    console.log(arr[i]);  
}
```



## 数组-多维数组

- 数组的元素可以是数组，可以多层嵌套，这就是多维数组：

```
var students = [  
    ['张三', 20],  
    ['李四', 21],  
    ['王五', 22],  
    ['小六', 23]  
];  
console.log(students.length);
```

## 数组-数组方法

- `join()`：将数组中的元素转换成字符串并连接在一起；默认是逗号(,)连接，可以指定连接符号：

```
var students = [  
    ['张三', 20],  
    ['李四', 21],  
    ['王五', 22],  
    ['小六', 23]  
];  
console.log(students.join());    //张三,20,李四,21,王五,22,小六,23  
console.log(students.join('|')); //张三,20|李四,21|王五,22|小六,23
```

## 数组-数组方法

- reverse() : 把数组中的元素顺序反转 :

```
var students = [  
    ['张三', 20],  
    ['李四', 21],  
    ['王五', 22],  
    ['小六', 23]  
];  
console.log(students.reverse().join('|')); //小六,23|王五,22|李四,21|张三,20  
console.log(students);                    //[['小六', 23], ['王五', 22], ['李四', 21], ['张三', 20]]
```

## 数组-数组方法

- `sort()`：对数组中的元素排序并返回排序后的值；

- 没有参数时按字母排序，会把元素转换成字符串：

```
var a = ['a', 'd', 'f', 'b', 'k', 'c'];  
a.sort();    //["a", "b", "c", "d", "f", "k"]
```

- 对元素是数字的数组进行排序时，是需要注意的：

- 下面的排序结果可能不是你想要的：

```
var a = [1, 23, 9, 12, 80];  
a.sort();    //[1, 12, 23, 80, 9];
```

- 也许下面的排序才是你想要的：

```
var a = [1, 23, 9, 12, 80];  
a.sort(function (a, b) {  
    return a-b; //返回负值，从小到大排序  
    return b-a; //返回正值，从大到小排序  
});  
                //[80, 23, 12, 9, 1]
```

## 数组-数组方法

- `concat()`：数组连接，连接后返回一个新数组；

- 实例：

```
var a = ['h', 'q', 'y', 'j'];  
console.log(a.concat('h5'));    // ["h", "q", "y", "j", "h5"]  
console.log(a = a.concat('javaee', '嵌入式'));    // ["h", "q", "y", "j", "javaee", "嵌入式"]  
console.log(a.concat(['AR/VR', 'Python']));    // ["h", "q", "y", "j", "javaee", "嵌入式", "AR/VR", "Python"]  
console.log(a.concat(['AR', 'VR'], ['Python', 'AI', '大数据', '自动化']));  
// ["h", "q", "y", "j", "javaee", "嵌入式", "AR", "VR", "Python", "AI", "大数据", "自动化"]
```

## 数组-数组方法

- slice()：截取数组，两个参数，指定开始和结束位置；
  - 参数支持负数，表示从后面往前面数，第一个-1，依次-2，-3...

- 实例：

```
var a = ['h', 'q', 'y', 'j'];  
console.log(a.slice(0, 2));    //["h", "q"]  
console.log(a.slice(2, 2));    //[]  
console.log(a.slice(2));      //["y", "j"]  
console.log(a.slice(-2));     //["y", "j"]  
console.log(a.slice(-2, -1));  //["y"]  
console.log(a.slice(-2, -3));  //[]
```

## 数组-数组方法

- splice(arg1, arg2, arg3, arg4, argn...) : 插入、删除、替换元素的通用方法 ;

- 会直接修改原数组;
- 参数1: 操作起始位置;
- 参数2: 操作元素个数;

为空表示到结尾; 为0表示插入元素; 大于0表示替换元素;

arg3, arg4, argn... : 新的元素, 代表要插入的元素;

- 实例 :

```
var a = ['h', 'q', 'y', 'j'];
```

```
a.splice(2);           //从下标2开始删除 ["h", "q"]
```

```
a.splice(2, 0, 1, 2, 3); //从下标2开始添加元素 ["h", "q", 1, 2, 3]
```

```
a.splice(2, 1, 'y', 'j'); //从下标2开始添加元素 ["h", "q", "y", "j", 2, 3]
```

```
a.splice(1, 0, ['H5', '嵌入式']); //["h", ["H5", "嵌入式"], "q", "y", "j", 2, 3]
```

# 数组-数组方法

- 数组元素的增删：
  - `push()`：在数组的结尾添加一个或多个元素，返回新数组的长度；
  - `pop()`：移除数组最后一个元素，返回移除的元素；
  - `unshift()`：在数组的开头添加一个或多个元素，返回新数组的长度；
  - `shift()`：删除数组第一个元素，并返回删除的元素；
  - 这四个函数都是在原始数组上进行修改；



## 数组-数组方法

- toString():将数组的元素转换成字符串，并用逗号连接起来：

```
var a = [  
    'h', 'q',  
    ['H5', 'JAVA'], {x:1, y:2}, 'y',  
    function () {console.log('函数');},  
    'j'  
];  
console.log(a.toString());  
//h,q,H5,JAVA,[object Object],y,function () {console.log('函数');},j  
  
console.log(a.join('|'));  
//h|q|H5, JAVA|[object Object]|y|function () {console.log('函数');}|j
```

## 数组-数组高级方法

- `forEach()`:遍历数组元素，并调用指定的函数；
  - `forEach`调用的函数支持三个参数，依次为元素、索引、数组本身；
  - `forEach`不支持`break`；

- 实例：

```
var a = ['h', 'q', 'y', 'j'];  
//打印每一个元素  
a.forEach(function (value) {  
    console.log(value);  
});  
//修改每一个元素的值  
a.forEach(function (value, i, arr) {  
    arr[i] = value + arr[i+1];  
});  
console.log(a);    //["hq", "qy", "yj", "jundefined"]
```

## 数组-数组高级方法

- `map()`:类似`forEach`，为每个元素调用指定函数，该函数一般有返回值；
  - 该操作返回一个新数组，不会修改原始数组；

- 实例：

```
var a = ['h', 'q', 'y', 'j'];  
var b = a.map(function (v) {  
    return v + v;  
});  
console.log(a);    // ["h", "q", "y", "j"]  
console.log(b);    // ["hh", "qq", "yy", "jj"]
```

## 数组-数组高级方法

- filter():返回数组的子集，是否返回当前元素是根据调用的函数来判断的;

- 调用的函数参数是数组的元素;

- 实例：

```
var a = [1,2,23,34,5,6,78,98,9];  
var oddnum = evennum = [];  
//得到数组里面的奇数  
oddnum = a.filter(function(v) { return v%2; });  
//得到数组里面的偶数  
evennum = a.filter(function(v) { return v%2 == 0; });  
console.log(oddnum);           // [1, 23, 5, 9]  
console.log(evennum);          // [2, 34, 6, 78, 98]
```

## 数组-数组高级方法

- every()和some():对数组的所有元素执行函数检查; 区别 :

- every(): 当每个元素都返回true是才返回true;
- some(): 当每个元素都返回false才返回false;

- 

- 实例 :

```
var a = [1,2,23,34,5,6,78,98,900];  
var oddnum = evennum = 0;  
//检查是不是都小于100  
oddnum = a.every(function(v) { return v < 100; });  
//至少有一个大于100  
evennum = a.some(function(v) { return v > 100; });  
console.log(oddnum);           //false  
console.log(evennum);          //true
```

## 数组-数组高级方法

- `reduce`和`reduceRight`：使用指定的函数对数组元素进行组合，生成一个值；
  - 参数1：要执行的函数，有返回值；
  - 参数2：传递给函数的默认值，可忽略；
  - `reduceRight`表示从右向左操作；

- 实例：

```
var a = [1,2,23,34,5,6,78,98,900];  
//求和，默认值为0  
//运算过程:0+1 = 1,然后1+1 = 2, 然后2+23 = 25, ...  
var n = a.reduce(function (x, y) { return x + y; }, 0);    // 1147  
//求积  
var m = a.reduce(function (x, y) { return x*y; }, 1);    // 322790832000  
//求最大值，可以不设置默认值  
var max = a.reduce(function (x, y) { return x>y?x:y; }); //900
```

## 数组-数组高级方法

- indexOf()和lastIndexOf：搜索指定的元素并返回其索引，不存在返回-1；
  - lastIndexOf：从后面往前搜索；
  - 参数1：要搜索的元素；
  - 参数2：从数组的哪个位置开始搜索，可省略，默认0；

- 实例：

```
var a = [1,2,23,34,5,6,78,98, 23,900];  
var n = a.indexOf(23);           //返回下标2  
var m = a.indexOf(23, 3);        //从下标3开始搜索，返回下标8  
var p = a.lastIndexOf(23);       //返回下标8
```

## 数组-数组类型

- `Array.isArray()` : 判断是否是数组 ;

- 实例 :

```
var a = [1,2,23,34,5,6,78,98, 23,900];  
//数组是特殊的对象类型  
console.log(typeof a);           //Object  
console.log(Array.isArray(a));   //true  
console.log(a instanceof Array); //true  
var b = {x:1, y:2};  
console.log(Array.isArray(b));   //false
```



## 数组-数组类型

- 数组是一种特殊类型的对象；
- 有些对象看起来像数组，称为类数组；

- 实例：

//类数组

```
function fn () {  
    console.log(arguments);  
    console.log(typeof arguments);  
    console.log(Array.isArray(arguments));  
}  
fn(1,2,3,4);  
//依次打印： Arguments(4)、 object、 false
```

# 华清远见-成都中心-H5教学部



# 目录

函数基本概念

函数的定义

嵌套函数

关键词this

形参和实参

# 目录

作为命名空间的函数

闭包

函数的劫持

高阶函数

# 函数-基本概念

- 函数是实现功能的代码块；
  - 一处定义，处处调用；
  - 如果把函数作为一个对象的属性，则称为方法；
  - 每次调用函数会产生一个this：谁调用这个函数或者方法，this就指向谁；
  - 函数就是对象，可以给他设置属性或方法；

# 函数-定义函数

- 使用function声明一个函数

- 语法结构:

```
function fn(形参){  
    函数体;  
    return 返回值;  
}
```

- 函数名后面跟上小括号()表示执行函数;

- 实例:

- 普通函数

```
function distance (x1, y1, x2, y2) {  
    var x = x2 - x1;  
    var y = y2 - y1;  
    return Math.sqrt(x*x + y*y).toFixed(2);  
}
```

# 函数-定义函数

- 递归函数实例:

```
function factorial (n) {  
    f(n==1) return 1;  
    return n*factorial(n-1);  
}
```

- 表达式函数实例:

```
var square = function (h) { return h*h; }  
console.log(square(3));
```

- 函数后面跟小括号表示直接执行该函数

```
var add = function (a,b) {  
    return a+b;  
}(23, 302);  
console.log(add);
```

# 函数-嵌套函数

- 函数是可以嵌套的；
  - 函数可以嵌套在其他函数里面，也就是在函数里面可以定义函数；
  - 被嵌套的函数可以访问父级函数的变量或参数；
  - 嵌套的函数不会从父级函数中继承this；如果想使用外层函数的this，需要把他保存到一个变量中；

- 实例：

```
function distance (x1, y1, x2, y2) {  
    function square(h) { return h*h;}  
    return Math.sqrt(square(x2-x1) + square(y2-y1)).toFixed(2);  
}
```



## 函数-函数调用

- 函数只有在调用时才会执行；
  - 作为函数；
  - 作为方法；
  - 作为构造函数；
  - 通过他们的call或者apply方法调用；
  - 函数名后面加小括号代表执行函数；

# 函数-形参和实参

- 函数的参数相对来讲还是比较自由；
  - 形参：在定义函数时使用的参数，用来接收调用函数时传进来的值；
  - 实参：是在调用时传递给函数的值；
  - 实参保存在一个类数组arguments里面，函数内部可用；
  - 如果实参少于形参个数，则多出的形参是undefined；
  - 如果实参多于形参，则只使用有效的实参，多出部分没影响；

- 实例：

```
function divide(a, b){  
    var b = b || 1;  
    return a/b;  
}  
console.log(divide(1));           //1  
console.log(divide(3,2,1));       //1.5
```

## 函数-作为值的函数

- 可以将函数作为值赋值给变量；也可以作为实参直接使用；

- 实例：

- 定义一个对象，其中一个属性值是函数：

```
var fn = { add:function (a, b) { return a + b; } };  
console.log(fn.add(3, 5));           //8
```

- 定义一个数组，其中一个元素是函数：

```
var arr = [function (a, b) {return a*b;}, 20, 30];  
console.log(arr[0](5, 20));          //100  
console.log(arr[0](arr[1], arr[2])); //600
```

# 函数-作为值的函数

- 深入理解:

//定义几个小函数

```
function add (a, b) {return a+b; }
```

```
function sub (a, b) {return a-b; }
```

```
function mul (a, b) {return a*b; }
```

```
function div (a, b) {return a/b; }
```

```
function allover (opname, arg1, arg2) {  
    return opname(arg1, arg2);  
}
```

//直接把函数作为实参传递进来

```
var n = allover(add, 2, 3);
```

```
console.log(n);           //5
```

//综合应用:  $= (6*8 - 10) + (20/5)$

```
var m = allover(add, sub(mul(6, 8), 10), div(20,5));
```

```
console.log(m);           //42
```

## 函数-给函数定义属性

- 给函数定义属性：

```
countIncrea.a = 0;  
function countIncrea () {  
    return ++countIncrea.a;  
}  
console.log(countIncrea ()); //1  
console.log(countIncrea ()); //2  
console.log(countIncrea ()); //3
```

## 函数-作为命名空间的函数

- 函数内声明的变量整个函数内可见，函数外不可见；
- 定义一个函数作为变量的作用域空间，这样不会污染到全局变量；避免同名变量冲突；
- 这在js里面是一种很常见的修改变量作用域的写法；
- 基本写法：

```
(function(){  
    //这里写你的代码  
})();
```

分析：定义一个匿名函数，函数后面跟上()表示直接执行该函数；

## 函数-闭包

- JS执行时用到作用域链，作用域链在函数定义时就已经定好了；每次执行函数时又有自己的新的作用域链；
- 函数内不仅包含函数主题，还要引用当前的作用域链；
- 函数体内部变量保存在函数作用域内，这种特性叫闭包；
- JS函数都是闭包：都是对象并关联到作用域链；
- 实例：

```
var school = '华清远见';  
function getschool () {  
    var school = '成都中心';  
    function getschool () { return school; }  
    return getschool;  
}  
var newschool = getschool();  
console.log(newschool);      //成都中心
```

## 函数-闭包

- 可以这么理解闭包：把外层函数看成对象，内部的局部变量看成属性，内部的函数看成对象的方法；
- 实例：

```
function counter (argument) {  
    var n = 0;  
    return {  
        count:function () { return n++; },  
        reset:function () { return n = 0; }  
    };  
}  
var c = counter();  
var d = counter();  
console.log(c.count());      //0  
console.log(d.count());      //0  
console.log(c.count());      //1  
console.log(c.reset());      //0  
console.log(d.count());      //1  
console.log(d.count());      //2  
console.log(c.count());      //0  
console.log(c.count());      //1
```



## 函数-函数的属性

- 函数是JavaScript代码中特殊的对象；
- length：属性，应该传递的实参个数，也就是形参个数；

```
function fn (a, b, c) {}  
console.log(fn.length);//3
```

- prototype：指向一个对象的引用，称为原型对象；  
console.log(Array.prototype);

## 函数-方法call和apply

- call()和apply()：实现方法劫持
- apply(为谁做事情, 一个数组参数)：
  - apply传递过来的数组参数会和方法的参数一一对应;
  - apply语法格式:被劫持的函数.apply();
- call(thisArg: any, args...: any)：
  - call传递的参数是一个一个的，这样可以很方便的和被劫持的函数参数保持一致;
- 函数被劫持以后会改变函数内this的指向；因为谁使用这个函数，this就指向谁；
- 实例：

```
var arr = [1, 80, -90, 501, 856];
```

```
var am = Math.max.apply(Array, arr);//得到最大值856
```

## 函数-方法call和apply

- 实例：自定义方法劫持实现继承概念

```
function Person(name, age){
    this.name = name;
    this.age = age;
    this.talk = function () {
        console.log(this.name + '的年龄是'+this.age+'岁');
    }
}

function itgirl (name, code, age) {
    Person.call(this, name, age);
    this.code = code;
    this.mycode = function () { console.log('我在学习' + this.code); }
}

var girl1 = new itgirl('一个女生', 'JAVA', 20);
girl1.talk();
girl1.mycode();
```

## 函数-方法toString

- toString：返回自定义函数的完整代码；

```
function fn (a, b, c) {'函数代码'}
```

```
var str = fn.toString();
```

```
console.log(str);
```

# 函数-高阶函数

- 高阶函数是指操作函数的函数；

- 实例：

```
function addnum (a, b, fn) { return fn(a) + fn(b); }  
var a = addnum(20, -30, Math.abs);           //50
```

- 实例：

```
function even (a) { return a%2 === 0; }  
//对函数判断进行反转  
function not (f) {  
    return function () {  
        var result = f.apply(this, arguments);  
        return !result;  
    }  
}  
var odd = not(even);  
console.log(odd);
```

# 华清远见-成都中心-H5教学部



# 目录

类和构造函数

用作命名空间的对象

全局对象Math

全局对象Date

全局对象String

# 类和对象-类和构造函数

- 构造函数用来创建实例对象；
  - 构造函数首字母大写；一般用于初始化一些属性值；

- 实例：

```
function Range (from, to) {  
    this.from = from;  
    this.to = to;  
}
```



# 类和对象-类和构造函数

- 构造函数的prototype属性用于定义该函数对象的原型；该原型会被所有的实例对象继承；
- 实例：

```
Range.prototype = {  
    foreach:function (f) {  
        for (var i = this.from; i <= this.to; i++) {f(i);}  
    },  
    avage:function () {  
        var n = 0;  
        for (var i=this.from; i<=this.to;i++) {n+= i;}  
        return (n/(this.to-this.from+1)).toFixed(2);  
    },  
    setfrom:function (from) {this.from = from;},  
    setto:function (to) {this.to = to;}  
};
```

# 类和对象-类和构造函数

- 使用关键词new调用构造函数创建一个实例对象;
- 实例:

```
var nr = new Range(2, 20);
```

- 新创建的实例对象会继承构造函数的原型, 也就是prototype属性;
- 实例:

```
nr.setfrom(10);  
nr.setto(15);  
nr.foreach(console.log);  
var avg = nr.avage();  
console.log(avg);
```

- 此时原型对象或实例都没有constructor(构造函数)属性, 需要显式指定: Range.prototype.constructor = Range;

# 类和对象-类的扩充

- JavaScript基于原型的继承机制是动态的；
  - 如果原型对象发生改变，继承这个原型的所有实例对象也跟着改变；
  - 实例：扩展构造函数原型

```
Range.prototype.sum = function () {  
    var sum = 0;  
    for (var i = this.from; i <= this.to; i++) {  
        sum += i;  
    }  
    return sum;  
}
```

- 已经实例化过的实例对象调用该扩展方法：nr.sum();

# 类和对象-类的扩充

- 内置构造函数扩展：

实例：

```
var numall = new Array(1, 56, 89, 901, 556);  
Array.prototype.sum = function () {  
    var total = 0;  
    for (var i = 0; i < this.length; i++) { total += this[i]; }  
    return total;  
}  
console.log(numall.sum());
```

- 禁止对构造函数扩展的方法：

- 设置为不可扩展：

```
Object.preventExtensions(Range.prototype);
```

- 设置为不可扩展不可删除：

```
Object.seal(Array.prototype);
```

# ■ 模块化编程-用作命名空间的对象

- 代码的模块化可以让代码在不同的场景中重复使用：
  - 模块化的中心思想是代码的可重用性；
  - 一般模块是一个独立的JS文件；里面可以是一个或一组相关的类、常用函数等
  - 模块是为了大规模JS开发；模块中尽可能少定义全局变量,最好不超过一个；
- 用作命名空间的对象：
  - 在模块创建过程中，用对象作为命名空间，可避免全局变量被污染；
  - 函数和变量可以作为命名空间的属性存储起来，而不是直接定义全局变量；

# ■ 模块化编程-用作命名空间的对象

- 实例：

- 创建一个JS文件，代码如下：

```
var set = {  
    Range:function(from, to) {  
        this.from = from; this.to = to;  
    }  
};  
set.Range.prototype = {  
    includes:function (a) {return a >= this.from && a <=this.to;}  
};  
//可以根据需要对原型进行扩展  
set.Range.prototype.foreach = function (f) {  
    for (var i = this.from; i <= this.to; i++) {f(i);}  
}
```

- 引入JS文件，使用该模块：

```
var nr = new set.Range(2, 20);  
console.log(nr.includes(3));
```

# 模块化编程-私有命名空间的函数

## • 用作私有命名空间的函数

- 在一个函数中定义的变量和函数是局部变量，也就是私有变量；可以使用函数的方式把成员变量变成私有的；
- 这里的函数一般是立即执行函数，也就是函数后面跟个小括号；
- 实例：

- 无参数：(function() { 函数体; })();
- 有参数：function(a, b) { 函数体; 使用参数a; 使用参数b; })(arg1, arg2);

- 参考：

```
var testmodel = (function () {  
    var n = 1;  
    function fn1 () { console.log('fn1') }  
    function fn2 () { console.log(n++); }  
    return {myfn1:fn1, myfn2:fn2};  
})();  
testmodel.myfn2();           //1  
testmodel.myfn1();           //fn1  
testmodel.myfn2();           //2
```

# 类和对象-全局对象-Math

- 数学函数和常量，没有构造函数；他的函数就是简单的函数，不是对象操作；

- 常量：

Math.E: 常量 $e$ ;

Math.PI: 常量 $\pi$ ;

- 静态函数：

Math.abs(x): 返回 $x$ 绝对值;

Math.ceil(x): 向上取整;

Math.floor(x): 向下取整;

Math.max(args...): 多个参数中的最大值;

Math.min(args...): 多个参数中的最小值;

Math.random(): 没有参数，返回 $0 \leq r < 1$ 的随机数 $r$ ;

Math.round(x): 四舍五入;

Math.pow(x, y): 计算 $x$ 的 $y$ 次方;

还有各种正余弦正余切自然对数等数学中常见的功能函数;



# | 类和对象-全局对象-Date

- Date对象用于操作日期和时间；

- 构造函数：

- new Date(): 不传参表示以当前时间创建一个Date对象；

- new Date(参数)：参数可以是一个时间戳或者时间格式的字符串，也可以多个参数，依次为年、月、日、时、分、秒、毫秒；

- 常用方法：

- getFullYear(): 返回四位数字的年份；

- getMonth(): 返回月份值0[一月]~11[十二月]；

- getDate(): 返回今天是多少号，1~31；

- getDay(): 返回星期几，0~6；

- getHours(): 返回小时；

- getMinutes(): 返回分钟；

- getSeconds(): 返回秒数；

- getMilliseconds(): 返回毫秒数；

- getTime(): 返回当前对象的时间戳；

# 类和对象-全局对象-String

- String对象用于操作字符串；
  - 属性：
    - length: 字符串长度;
  - 常用方法：

|                |                |
|----------------|----------------|
| charAt():      | 返回指定位置的字符;     |
| concat():      | 字符串连接操作;       |
| indexOf():     | 检索字符串;         |
| lastIndexOf(): | 反方向检索字符串;      |
| trim():        | 去除字符串前后的空格;    |
| split():       | 字符串分割成数组;      |
| replace():     | 使用正则表达式替换子字符串; |
| Search():      | 查找子字符串;        |
| substr():      | 截取字符串;         |
| toLowerCase(): | 字符串转小写;        |
| toUpperCase(): | 字符串转大写;        |

# 第24章 正则表达式

华清远见-成都中心-H5教学部



# 目录

正则表达式基本概念

直接量字符

字符类

重复字符

选择和分组

# 目录

指定位置

修饰符

String支持正则表达式的方法

正则表达式对象的方法

# 正则表达式-基本概念及定义方式

- 正则表达式：regular expression，是描述字符模式的对象；
- JavaScript中的RegExp类表示正则表达式；
- 正则表达式的定义：

- 直接包含在一对斜杠 (/) 之间；

```
var pattern = /^hcyj/;
```

- 创建一个新的RegExp对象；

```
var pattern = new RegExp("^hcyj");
```

# | 正则表达式-直接量字符

- 直接量字符；
  - 正则表达式中的字母和数字按字面含义进行匹配；
  - 支持非字母的字符匹配，而这些字符就需要反斜线(\)进行转义了；
  - 常见的直接量字符：

| 字符     | 匹配                     |
|--------|------------------------|
| -----  |                        |
| 字母和数字  | 匹配自身                   |
| \t     | 制表符                    |
| \n     | 换行符                    |
| \r     | 回车符                    |
| \uxxxx | 由十六进制数xxxx指定的Unicode字符 |

# | 正则表达式-字符类

- 字符类是指代表特殊含义而不再是字面意思；
  - 如果需要匹配字符类的字面意思，需要反斜线(\)进行转义；
  - 常见的正则表达式字符类：

| 字符     | 匹配                           |
|--------|------------------------------|
| -----  |                              |
| [...]  | 方括号内的任意字符                    |
| [^...] | 不在方括号内的任意字符                  |
| .      | 除换行符和其他行终止符之外的任意字符           |
| \w     | 任何ASCII字符单词，等价于[a-zA-Z0-9]   |
| \W     | 任何非ASCII字符单词，等价于[^a-zA-Z0-9] |
| \s     | 任何Unicode空白符                 |
| \S     | 任何非Unicode空白符，包括各种特殊字符       |
| \d     | 任何ASCII数字，等价于[0-9]           |
| \D     | ASCII数字之外的任何字符，等价于[^0-9]     |



# | 正则表达式-重复字符

- 有时候需要对指定字符多次匹配；
  - 可以在正则表达式中定义元素的重复出现次数；
  - 正则表达式重复字符语法：

| 字符     | 匹配                   |
|--------|----------------------|
| <hr/>  |                      |
| {n, m} | 匹配前一项至少n次，不能超出m次     |
| {n, }  | 匹配前一项至少n次，或者更多次      |
| {n}    | 匹配前一项n次              |
| ?      | 匹配前一项0或1次，等价于 {0, 1} |
| +      | 匹配前一项1或者多次，等价于 {1, } |
| *      | 匹配前一项0或者多次，等价于 {0, } |

# | 正则表达式-选择和分组

- 正则表达式支持选择项或子表达式；

- 选择和分组：

字符

含义

|

选择，匹配的是左边或右边子表达式

(...)

组合，将几个选项组合到一个单元

# | 正则表达式-指定位置

- 指定匹配位置；

- 指定位置：

字符

含义

-----  
^

匹配字符串的开头

\$

匹配字符串的结尾

\b

匹配单词的边界，但是不包含空格

(?=p)

零宽正向先行断言，要求接下来的字符与p匹配，但不能包括匹配p的那些字符

(?!p)

零宽负向先行断言，要求接下来的字符不与p匹配

- 零宽断言：我们说断言一般是指零宽断言，它匹配到的内容不会保存到匹配结果中去，只是用于指定一个位置而已。

# | 正则表达式-修饰符

- 正则表达式支持修饰符；
  - 修饰符放在正则表达式的后边界之后：

| 字符 | 含义                  |
|----|---------------------|
| i  | 不区分大小写              |
| g  | 全部匹配                |
| m  | 多行匹配模式，^表示开始，\$表示结束 |

# String支持正则表达式的方法

- 支持正则表达式的String对象方法：

- `search(regex)`:
  - 参数是正则表达式，返回第一个匹配的子串位置，找不着返回-1；
  - 因为只返回第一个匹配子串，所以忽略修饰符g；
  - 如果参数不是正则表达式则转成正则表达式；
- `replace(var1, var2)`:
  - 第一个参数是正则表达式，第二个参数是要替换成的字符串；
  - 如果参数一不是正则表达式，则直接把对应的字符串替换为参数二；
  - 如果修饰符有g，会全部替换，否则只替换第一个；
- `match(regex)`:
  - 参数是正则表达式，返回匹配的结果，数组；
  - 如果有修饰符g则全部匹配，否则只匹配你第一个；
- `split()`:
  - 用指定字符串把字符串分割成数组；
  - 参数支持字符串和正则表达式；

# 正则表达式对象的方法

- 正则表达式的方法；
  - `exec(str)`:
    - 参数是字符串，对该字符串进行匹配检索；
    - 总是返回一个结果，并提供匹配的完整信息；
    - 再次执行会进行下次匹配；
    - 没找到返回`null`；
  - `test(str)`:
    - 对字符串进行检测，包含返回`true`，否则返回`false`；

# 华清远见-成都中心-H5教学部



# 目录

ES6基本概念

ES6兼容问题

变量的定义及其作用域

常量的定义及不可改变性

解构赋值



# 目录

字符串操作

数组操作

函数扩展

类

模块化

# 目录

Promise

Generator

async

# ES6-基本概念

- ES6是个什么东西？

- ES:ECMAScript, js语法规范；
- ECMAScript 6.0（简称ES6）是JavaScript语言的最新标准，已在2015年6月正式发布，目标是使JS可以用来编写复杂的大型应用程序，成为企业级开发语言。
- “ES6”就是指JavaScript语言的当前最新版本；
- ES6是一个历史名词，也是一个泛指，含义是5.1版以后的下一代标准，涵盖了ES2015、ES2016、ES2017、ES2018；

# | ES6-兼容问题

- 兼容问题

- 当前浏览器对ES6的支持已经很好了，但是还是存在一定的兼容性问题；
- 为了让代码更好的运行在所有浏览器下，我们可以使用Babel对ES6进行转码，转换成ES5语法，所以我们可以放心的使用ES6写代码，而不用担心兼容问题。
- ES6作为新一代标准推进的非常顺畅，浏览器在不断的完善对他的支持，很快我们将不需要转码；

# | ES6-变量的定义

- ES6变量的定义

- ES6里面新增了let来声明变量;

```
let a= 10;
```

```
let b = 20;
```

```
console.log(a, b);
```

- let声明过的变量不能重复声明;

```
let a = 10;
```

```
let a = 20; //报错: Identifier 'a' has already been declared
```

## ES6-变量作用域

- let声明的变量：
  - ES6支持块级作用域；
  - 用let声明的变量的作用域只在当前代码块里面，外面不可以使用；
  - 不会污染全局变量，其他地方也有可能定义了这个变量，互不影响；
  - 不存在变量提升：按照逻辑，变量应该在声明之后才可以使用；
  - 实例：

```
{  
    console.log(a); //Uncaught ReferenceError: a is not defined  
    let a = 20;  
    console.log(a);  
    var h = 'hqyj' ;  
}  
console.log(h);  
console.log(a);           //报错： a is not defined
```

## ES6-变量作用域

- 块级作用域的应用：

- 用于循环，防止循环的变量泄漏为全局变量；
- 不再需要立即执行函数来防止变量污染；
- 实例：

// 立即执行函数写法

```
(function () {  
    var hqyj = ‘华清远见’;  
})();
```

//块级作用域写法

```
{  
    let hqyj = ‘华清远见’;  
}
```

# ES6-常量的定义

- 常量的定义；
  - 使用const定义常量，一旦定义，不可修改；
  - 同样也不能重复声明；
  - 在定义常量的时候必须给初始值，因为以后再也无法赋值了；
  - 常量的好处：见名知意、一处修改处处修改；
  - 实例：

```
const a = '华清远见';  
console.log(a);  
a = 10;//报错： Assignment to constant variable
```



# | ES6-常量的不可变

- 常量的不可变：

- 常量的不可变实际上是指向的内存地址存储的数据不能修改；
- 如果只是基础变量类型，值就在内存地址里面，是不可改变的；
- 对于复合类型的数据对象和数组来讲，里面存储的是引用地址，只能保证引用地址不变，但是引用地址指向的数据是否改变就不可控；
- 实例：

```
//不能改变地址引用
```

```
const obj = {username:'华沫'};
```

```
obj.age = 20;
```

```
obj.username = '华清';
```

```
console.log(obj);
```

# ES6-解构赋值

- 解构赋值：

- 一次定义多个变量，并批量赋值；

- 实例：

```
let [a, b, c] = [1, 2, 3];
```

```
console.log(a, b, c);
```

- 对象方式的解构赋值是根据键值赋值的，跟顺序无关；

- 实例：

```
let {d, e, f} = {f:30, d:10, e:20}
```

```
console.log(f);
```

- 如果赋值失败，则返回值是undefined；

- 支持设置默认值：

```
let {h='华', q='清', y, j} = {y:'远', j:'见'};
```

```
console.log(h,q,y,j);
```

# ES6-解构赋值

- 解构赋值：

- 支持多层嵌套的解构赋值：

```
[{a, b}, c] = [{a:1, b:9}, 8];
```

```
console.log(a,b,c);
```

- 作用：

- 变量交换；
  - 方便函数返回多个值；
  - 方便函数定义形参及默认值；

# ES6-字符串操作

- 字符串模板：

- 使用反引号(`)把子串引起来，需要替换的地方用\${变量名}；
- 实例：

```
let name = '华仔'；
```

```
let tel = '15982036532'；
```

```
let course = 'H5'；
```

```
// ` 不是单引号，是反引号`
```

```
let str1 = `免费试听线上VIP课程，姓名【${name}】+【${tel}】帮您申请创客学院学习卡，  
可以免费观看${course}教程，  
学习卡卡号密码会以在线形式发送给您`；
```

```
console.log(str1)；
```

# | ES6-字符串操作

- 循环字符串：

- 使用for...of循环字符串：  
for (let c of 'hqyj') { console.log(c); }

- 可以代替indexOf的方便好用的方法：

- includes(): 返回布尔值，检查字符串是否包含指定字符串；
- startsWith(): 返回布尔值，检查字符串是否以指定字符串开头；
- endsWith(): 返回布尔值，检查字符串是否以指定字符串结尾；
- 实例：

```
let str = '华清远见成都中心';  
console.log(str.includes('成都')); //true  
console.log(str.startsWith('华清')); //true  
console.log(str.endsWith('中心')); //true
```

## ES6-字符串操作

- 新增字符串加工处理方法：

- repeat(n): 将指定字符串重复n次;
- padStart(n, str): 使用str从开始位置将字符串填充到指定长度n;
- padEnd(n, str): 从结尾位置开始使用str将字符串填充到指定长度n;
- 实例:

```
let str = '成都中心';  
console.log(str.repeat(3));           //成都中心成都中心成都中心'  
console.log(str.padStart( 8,'华清远见')); //华清远见成都中心  
console.log(str.padEnd(7, '教学部'));  //成都中心教学部
```

# | ES6-数组操作

- 数组操作

- 扩展运算符(...): 将一个数组转为用逗号分隔的序列;

- 方便的数组复制:

```
var arr1 = [1, 2, 3, 4, 5];
```

```
var arr3 = [...arr1];
```

- 方便的函数劫持

```
// 使用扩展运算符
```

```
Math.max(...[1, 11, 2]);
```

- 方便的实现数组合并、解构赋值、字符串转数组等相关操作;

# | ES6-数组循环

- for...of循环

- for...of循环出来的直接就是值；支持数组，不支持对象类型；  
var arr1 = [1, 2, 3, 4, 5];
- 循环下标：  
for (var x in arr1) { console.log(x); }
- 直接循环值  
for(var x of arr1){ console.log(x); }
- 同时循环值和下标  
for(var x of arr1.entries()){console.log(x); }
- 循环下标  
for(var x of arr1.keys()){ console.log(x); }



# | ES6-箭头函数

- 给参数设置默认值

- 可以直接给参数设置默认值:

```
function add(a=0, b=0){  
    return a + b  
}
```

```
console.log(add(1));
```

- 需要设置默认值的参数应该都放在最后面;
  - 函数的length属性返回的是没有指定默认值的参数个数;

# | ES6-箭头函数

- rest参数

- ES6 支持rest 参数，定义方式：(...变量名);
- rest是一个数组，接收传递给函数除形参之外的所有的实参;
- 实例:

```
function add(...nums){  
    let a = 0;  
    for(let i of nums) a += i;  
    return a;  
}  
console.log(add(1,2,3)); //6
```

```
function fn(a, ...rest){  
    return rest;  
}  
console.log(fn(1,2,3));  //[2,3]
```

# | ES6-箭头函数

- 箭头函数

- 箭头函数定义的语法结构:

函数名称=(形参)=>{ 函数体; }

- 实例:

```
showname=(myname)=>{ console.log(myname); }
```

```
showname('学生1');
```

- 箭头函数里面的this指向window;

```
var obj = document.querySelector('.cl');
```

```
obj.onclick = ()=>{
```

```
    console.log(this);
```

```
    //这里的this指向window
```

```
    obj.style.backgroundColor = 'red';
```

```
}
```

# | ES6-函数

- 箭头函数

- 箭头函数没有arguments，你可以使用reset；
- 不可以当做构造函数；
- 箭头函数里面的this指向window；

```
var obj = document.querySelector('.cl');  
obj.onclick = ()=>{  
    console.log(this);  
    //这里的this指向window  
    obj.style.backgroundColor = 'red';  
}
```

## | ES6-类

- 在ES6中，class (类)作为对象的模板被引入，可以通过 class 关键字定义类。
- class 的本质是 function。
- 它可以看作一个语法糖，让对象原型的写法更加清晰、更像面向对象编程的语法。
- 类不可重复声明
- 类定义不会被提升，这意味着必须在访问前对类进行定义，否则就会报错

## | ES6-类

- 类声明时使用首字母大写（驼峰法）
- ES6 的类，完全可以看作构造函数的另一种写法。

```
class Point {  
  // ...  
}
```

```
typeof Point // "function"
```

```
Point === Point.prototype.constructor // true
```

- 类的数据类型就是函数，类本身就指向构造函数。

## ES6-类

- 使用的时候，也是直接对类使用new命令，跟构造函数的用法完全一致。

```
class Bar {  
  doStuff() {  
    console.log('stuff');  
  }  
}
```

```
var b = new Bar();  
b.doStuff()
```

## | ES6-类

- 构造函数的prototype属性，在 ES6 的“类”上面继续存在。事实上，类的所有方法都定义在类的prototype属性上面。

```
class Point {  
  constructor() {  
    // ...  
  }  
  toString() {  
    // ...  
  }  
  toValue() {  
    // ...  
  }  
}
```

// 等同于

```
Point.prototype = {  
  constructor() {},  
  toString() {},  
  toValue() {},  
};
```



## ES6-类

- 在类的实例上面调用方法，其实就是调用原型上的方法。
- 由于类的方法都定义在prototype对象上面，所以类的新方法可以添加在prototype对象上面。Object.assign方法可以很方便地一次向类添加多个方法。
- 类的内部所有定义的方法，都是不可枚举的（non-enumerable）。

```
class Point {  
  constructor(){  
    // ...  
  }  
}
```

```
Object.assign(Point.prototype, {  
  toString(){},  
  toValue(){  
  }});
```

## | ES6-类

- constructor方法是类的默认方法，通过new命令生成对象实例时，自动调用该方法。一个类必须有constructor方法，如果没有显式定义，一个空的constructor方法会被默认添加。支持构造函数constructor，实例化对象时，构造函数默认执行；

```
class Point {  
}
```

// 等同于

```
class Point {  
  constructor() {}  
}
```

## ES6-类

- constructor方法默认返回实例对象（即this），完全可以指定返回另外一个对象。

```
class Foo {  
  constructor() {  
    return Object.create(null);    //constructor函数返回一个全新的对象  
  }  
}
```

```
new Foo() instanceof Foo //false
```

## ES6-类

- 与 ES5 一样，在“类”的内部可以使用get和set关键字，对某个属性设置存值函数和取值函数，拦截该属性的存取行为。

```
class MyClass {  
  constructor() {  
    // ...  
  }  
  get prop() {  
    return 'getter';  
  }  
  set prop(value) {  
    console.log('setter: ' + value);  
  }  
}
```

```
let inst = new MyClass();
```

```
inst.prop = 123;  
// setter: 123
```

```
inst.prop  
// 'getter'
```

## | ES6-类

- 类的方法内部如果含有this，它默认指向类的实例。
- 如果单独使用该方法，那么this的指向会发生改变，解决办法是：
  - 为这个方法绑定this，如：`someFunc().bind(this)`
  - 使用箭头函数：箭头函数内部的this总是指向定义时所在的对象

## ES6-类

- 类相当于实例的原型，所有在类中定义的方法，都会被实例继承。如果在一个方法前，加上static关键字，就表示该方法不会被实例继承，而是直接通过类来调用，这就称为“静态方法”。
- 静态方法只能通过类名调用

```
class Foo {  
  static classMethod() {  
    return 'hello';  
  }  
}
```

```
Foo.classMethod() // 'hello'
```

```
var foo = new Foo();
```

```
foo.classMethod() // TypeError: foo.classMethod is not a function
```

## ES6-类

- 如果静态方法包含this关键字，这个this指的是类，而不是实例。

```
class Foo {  
  static bar() {  
    this.baz();  
  }  
  static baz() {  
    console.log('hello');  
  }  
  baz() {  
    console.log('world');  
  }  
}
```

```
Foo.bar() // hello
```

```
var foo = new Foo();  
foo.baz() //world
```

## ES6-类

- 父类的静态方法，可以被子类继承

```
class Foo {  
  static classMethod() {  
    return 'hello';  
  }  
}
```

```
class Bar extends Foo {  
}
```

```
Bar.classMethod() // 'hello'
```



## ES6-类

- 实例属性除了定义在constructor()方法里面的this上面，也可以定义在类的最顶层。
- 这种新写法的好处是，所有实例对象自身的属性都定义在类的头部，看上去比较整齐，一眼就能看出这个类有哪些实例属性。

```
class IncreasingCounter {  
  _count = 0;    //实例属性  
  bar = 'hello';  
  baz = 'world';  
  
  get value() {  
    return this._count;  
  }  
  increment() {  
    this._count++;  
  }  
}
```

## ES6-类的继承

- 解决代码的复用
- 使用**extends**关键字实现继承
- 子类可以继承父类中所有的方法和属性
- 子类只能继承一个父类（**单继承**），一个父类可以有多个子类
- 子类的构造方法中必须有super()来指定调用父类的构造方法，并且位于子类构造方法中的第一行
- 子类中如果有与父类相同的方法和属性，将会优先使用子类的（覆盖）

## ES6-类的继承

- 如果子类没有定义constructor方法，这个方法会被默认添加，代码如下。也就是说，不管有没有显式定义，任何一个子类都有constructor方法。

```
class ColorPoint extends Point {  
}
```

// 等同于

```
class ColorPoint extends Point {  
  constructor(...args) {  
    super(...args);  
  }  
}
```

## ES6-类的继承

- 在子类的构造函数中，只有调用super之后，才可以使用this关键字，否则会报错。这是因为子类实例的构建，基于父类实例，只有super方法才能调用父类实例。

```
class Point {  
  constructor(x, y) {  
    this.x = x;  
    this.y = y;  
  }  
}  
  
class ColorPoint extends Point {  
  constructor(x, y, color) {  
    this.color = color; // ReferenceError  
    super(x, y);  
    this.color = color; // 正确  
  }  
}
```

## ES6-类的继承

- `super`关键字，既可以当作函数使用，也可以当作对象使用。在这两种情况下，它的用法完全不同。
  - `super`作为函数调用时，代表父类的构造函数。ES6 要求，子类的构造函数必须执行一次`super`函数。只能用在子类的构造函数之中

```
class B extends A {  
  constructor() {  
    super();  
  }  
}
```

## ES6-类的继承

- `super`作为对象时，在普通方法中，指向父类的原型对象；在静态方法中，指向父类。

```
class A {  
  p() {  
    return 2;  
  }  
}
```

```
class B extends A {  
  constructor() {  
    super();  
    console.log(super.p()); // 2  
  }  
}
```

## ES6-类的继承

- 由于super指向父类的原型对象，所以定义在父类实例上的方法或属性，是无法通过super调用的。

```
class A {  
  constructor() {  
    this.p = 2;  
  }  
}
```

```
class B extends A {  
  get m() {  
    return super.p;  
  }  
}
```

```
let b = new B();  
b.m // undefined
```

- 解决办法是：将属性定义在父类的原型上：  
A.prototype.x = 2;

# ES6-模块化

- 概述：
  - ES6 引入了模块化，其设计思想是在编译时就能确定模块的依赖关系，以及输入和输出的变量。
  - ES6 的模块化分为导出（**export**） 与导入（**import**）两个模块。



# ES6-模块化

- 特点：

- ES6 的模块自动开启严格模式，不管你有没有在模块头部加上 `use strict`;
- 模块中可以导入和导出**各种类型**的变量，如函数，对象，字符串，数字，布尔值，类等
- 每个模块都有自己的上下文，每一个模块内声明的变量都是局部变量，不会污染全局作用域
- 每一个模块只加载一次（是单例的），若再去加载同目录下同文件，直接从内存中读取

## ES6-模块化

- export导出：与default关联使用，并且一个js模块中只能有一个export default语句

- 导出字符串

```
export default "abc";
```

- 导出数字

```
export default 123;
```

- 导出布尔值

```
export default true;
```

- 导出数组

```
export default [1,2,3];
```

# | ES6-模块化

- 导出对象

```
var obj = {  
    name: '张三',  
    age: 20  
}  
  
export default obj;
```

- 导出函数

```
var func = function() {  
    console.log("func函数");  
    return 100;  
}  
  
export default func;
```

# | ES6-模块化

- 导出类

```
class People {  
  constructor() {  
    this.a = 100;  
  }  
  say() {  
    console.log("say...");  
  }  
}  
  
export default People;
```

# | ES6-模块化

- import导入

- 与from关联使用，此时script标签的type必须设置为module
- 单例模式：多次重复执行同一句 import 语句，那么只会执行一次，而不会执行多次。import 同一模块，声明不同接口引用，会声明对应变量，但只执行一次 import

```
<script type="module">  
  import People from './js/myModule.js';  
  let p = new People();  
  p.say();  
</script>
```

## | ES6-Promise

- Promise 是异步编程的一种解决方案，比传统的解决方案——回调函数和事件——更合理和更强大。
- 所谓Promise，简单说就是一个容器，里面保存着某个未来才会结束的事件（通常是一个异步操作）的结果。
- 从语法上说，Promise 是一个对象，从它可以获取异步操作的消息。
- Promise 提供统一的 API，各种异步操作都可以用同样的方法进行处理。

## ES6-Promise状态

- Promise 异步操作有三种状态：
  - pending: 初始状态，不是成功或失败状态。
  - fulfilled: 意味着操作成功完成。
  - rejected: 意味着操作失败
- 除了异步操作的结果，任何其他操作都无法改变这个状态。
- Promise 对象的状态改变只有：
  - 从 pending 变为 fulfilled
  - 从 pending 变为 rejected
  - 只要处于 fulfilled 和 rejected，状态就不会再变了即 resolved（已定型）。

## ES6-Promise优缺点

- 优点：

- 可以将异步操作以同步操作的流程表达出来，避免了层层嵌套的回调函数。
- Promise 对象提供统一的接口，使得控制异步操作更加容易。

- 缺点：

- 无法取消Promise，一旦新建它就会立即执行，无法中途取消。
- 如果不设置回调函数，Promise内部抛出的错误，不会反应到外部。
- 当处于pending状态时，无法得知目前进展到哪一个阶段（刚刚开始还是即将完成）。



## ES6-Promise创建

- ES6 规定，Promise对象是一个构造函数，用来生成Promise实例。使用**new**关键字

```
const promise = new Promise(function(resolve, reject) {  
  // ... some code  
  
  if (/* 异步操作成功 */) {  
    resolve(value);  
  } else {  
    reject(error);  
  }  
});
```

- Promise构造函数接受一个函数作为参数，该函数的两个参数分别是resolve和reject。

## ES6-Promise对象参数

- resolve函数的作用是，将Promise对象的状态从“未完成”变为“成功”（即从 pending 变为 resolved），在异步操作成功时调用，并将异步操作的结果，作为参数传递出去。
- reject函数的作用是，将Promise对象的状态从“未完成”变为“失败”（即从 pending 变为 rejected），在异步操作失败时调用，并将异步操作报出的错误，作为参数传递出去。

## ES6-Promise-then方法

- Promise实例生成以后，可以用then方法分别指定resolved状态和rejected状态的回调函数。

```
promise.then(function(value) {  
    // success 对应resolved状态  
}, function(error) {  
    // failure 对应rejected状态  
});
```

- then方法可以接受两个回调函数作为参数
  - 第一个回调函数是Promise对象的状态变为resolved时调用
  - 第二个回调函数是Promise对象的状态变为rejected时调用（可选，不一定要提供）
  - 这两个函数都接受Promise对象传出的值作为参数。

## ES6-Promise-then方法链式调用

- 从表面上看，Promise只是能够简化层层回调的写法，而实质上，Promise的精髓是“状态”，用维护状态、传递状态的方式来使得回调函数能够及时调用，它比传递callback函数要简单、灵活的多。所以使用Promise的正确场景是这样的：

```
runAsync1()  
  .then(function(data) {  
    console.log(data);  
    return runAsync2();  
  })  
  .then(function(data) {  
    console.log(data);  
    return runAsync3();  
  })  
  .then(function(data) {  
    console.log(data);  
  })  
  );
```

## ES6-Promise-then方法链式调用

- 上面的代码中：runAsync1()，runAsync2()，runAsync3()是3个异步任务，分别返回一个Promise对象，通过在then中返回一个对象，可以在下一个then中接收该任务的状态。这种链式调用可以按顺序执行异步任务。

- 在链式调用中，then可以返回一个Promise对象，也可以返回一个数据，那么在后面的then中直接接收数据

```
runAsync1()  
  .then(function(data) {  
    console.log(data);  
    return '直接返回数据'; //这里直接返回数据  
  })  
  .then(function(data) {  
    console.log(data); //接收上一个then中return的数据  
  });
```

## ES6-Promise-catch方法

- Promise对象除了then方法，还有一个catch方法，它是做什么用的呢？其实它和then的第二个参数一样，用来指定reject的回调，用法是这样：

```
runAsync1()  
  .then(function(data) {  
    console.log('resolved');  
    console.log(data);  
  }).catch(function(reason) {  
    console.log('rejected');  
    console.log(reason);  
  });
```

- 如果then结构中抛出异常了，不会报错卡死JS，而是会进到这个catch方法中。

# | ES6-Promise-ajax

- 使用Promise实现Ajax

```
const getJSON = function(url) {  
  const promise = new Promise(function(resolve, reject) {  
    const handler = function() {  
      if (this.readyState !== 4) {  
        return;  
      }  
      if (this.status === 200) {  
        resolve(this.response);  
      } else {  
        reject(new Error(this.statusText));  
      }  
    };  
    const client = new XMLHttpRequest();  
    client.open("GET", url);  
    client.onreadystatechange = handler;  
    client.responseType = "json";  
    client.setRequestHeader("Accept", "application/json");  
    client.send();  
  });  
  return promise;  
};
```

## | ES6-Promise-ajax

```
getJSON("/posts.json")  
  .then(function(json) {  
    console.log('Contents: ' + json);  
  })  
  .catch(function(reason) {  
    console.error('出错了', error);  
  })
```



# | ES6-Generator

- ES6 新引入了 Generator 函数，可以通过 yield 关键字，把函数的执行流挂起，为改变执行流程提供了可能，从而为异步编程提供解决方案。
- 形式上，Generator 函数是一个普通函数，但是有两个特征：
  - 在 function 后面，函数名之前有个 \*
  - 函数内部使用 yield 表达式，定义不同的内部状态（yield在英语里的意思就是“产出”）其中 \* 用来表示函数为 Generator 函数，yield 用来定义函数内部的状态。

# | ES6-Generator

- Generator函数定义示例：

```
function* helloWorldGenerator() {  
  yield 'hello';  
  yield 'world';  
  return 'ending';  
}  
  
var hw = helloWorldGenerator();
```

- 上面代码定义了一个 Generator 函数helloWorldGenerator，它内部有两个yield表达式（hello和world），即该函数有三个状态：hello，world 和 return 语句（结束执行）。

## | ES6-Generator

- Generator 函数的调用方法与普通函数一样，也是在函数名后面加上一对圆括号。不同的是，调用 Generator 函数后，该函数并不执行，返回的也不是函数运行结果，而是一个指向内部状态的指针对象。
- 所以必须调用遍历器对象的**next**方法，使得指针移向下一个状态。也就是说，每次调用next方法，内部指针就从函数头部或上一次停下来的地方开始执行，直到遇到下一个yield表达式（或return语句）为止。
- 换言之，Generator 函数是分段执行的，yield表达式是暂停执行的标记，而next方法可以恢复执行。

# | ES6-Generator

- Generator函数调用示例：

```
hw.next()    //第一次调用  
// { value: 'hello', done: false }
```

```
hw.next()    //第二次调用  
// { value: 'world', done: false }
```

```
hw.next()    //第三次调用  
// { value: 'ending', done: true }
```

```
hw.next()    //第四次调用  
// { value: undefined, done: true }
```

# | ES6-Generator

- 调用 Generator 函数，返回一个遍历器对象，代表 Generator 函数的内部指针。
- 每次调用遍历器对象的next方法，就会返回一个有着value和done两个属性的对象。
  - value属性表示当前的内部状态的值，是yield表达式后面那个表达式的值；
  - done属性是一个布尔值，表示是否遍历结束。
- 当返回的对象中value的值为undefined，done的值为true时，表示这个Generator函数执行完毕

## | ES6-Generator

- ES6 没有规定，function关键字与函数名之间的星号，写在哪个位置。这导致下面的写法都能通过。
  - `function * foo(x, y) { ... }`
  - `function *foo(x, y) { ... }`
  - `function* foo(x, y) { ... }`
  - `function*foo(x, y) { ... }`
- 由于 Generator 函数仍然是普通函数，所以一般的写法是上面的第三种，即星号紧跟在function关键字后面。

## ES6-Generator 函数的this

- Generator 函数总是返回一个遍历器，ES6 规定这个遍历器是 Generator 函数的实例，也继承了 Generator 函数的prototype对象上的方法。
- Generator函数总是返回一个遍历器对象，无法返回this，所以Generator函数不能当成构造函数使用，也就无法使用new关键字。

```
function* F() {  
  yield this.x = 2;  
  yield this.y = 3;  
}
```

```
new F()
```

```
// TypeError: F is not a constructor
```

## ES6-Generator 函数的this

- 那么，有没有办法让 Generator 函数返回一个正常的对象实例，既可以用next方法，又可以获得正常的this？
- 首先，生成一个空对象，使用call方法绑定 Generator 函数内部的this。这样，构造函数调用以后，这个空对象就是 Generator 函数的实例对象了。

```
function* F() {  
  this.a = 1;  
  yield this.b = 2;  
  yield this.c = 3;  
}  
var obj = {};  
var f = F.call(obj);
```

```
f.next(); // Object {value: 2, done: false}  
f.next(); // Object {value: 3, done: false}  
f.next(); // Object {value: undefined, done: true}
```

```
obj.a // 1  
obj.b // 2  
obj.c // 3
```



## ES6-Generator 函数的this

- 上面代码中，执行的是遍历器对象f，但是生成的对象实例是obj，有没有办法将这两个对象统一呢？
- 答：**将obj换成F.prototype**

```
function* F() {  
  this.a = 1;  
  yield this.b = 2;  
  yield this.c = 3;  
}  
var f = F.call(F.prototype);
```

```
f.next(); // Object {value: 2, done: false}  
f.next(); // Object {value: 3, done: false}  
f.next(); // Object {value: undefined, done: true}
```

```
f.a // 1  
f.b // 2  
f.c // 3
```

## ES6-Generator 函数的this

- 再将F改成构造函数，就可以对它执行new命令了

```
function* gen() {  
  this.a = 1;  
  yield this.b = 2;  
  yield this.c = 3;  
}
```

```
function F() {  
  return gen.call(gen.prototype);  
}
```

```
var f = new F();
```

```
f.next(); // Object {value: 2, done: false}  
f.next(); // Object {value: 3, done: false}  
f.next(); // Object {value: undefined, done: true}
```

```
f.a // 1  
f.b // 2  
f.c // 3
```

# ES6-Generator 应用

## • (1) 异步操作的同步化表达

- Generator 函数的暂停执行的效果，意味着可以把异步操作写在yield表达式里面，等到调用next方法时再往后执行。
- 这实际上等同于不需要写回调函数了，因为异步操作的后续操作可以放在yield表达式下面，反正要等到调用next方法时再执行。
- 所以，Generator 函数的一个重要实际意义就是用来处理异步操作，改写回调函数。

## ES6-Generator 应用

- Ajax 是典型的异步操作，通过 Generator 函数部署 Ajax 操作，可以用同步的方式表达。

```
function* main() {  
  var result = yield request("http://some.url");  
  var resp = JSON.parse(result);  
  console.log(resp.value);  
}
```

```
function request(url) {  
  makeAjaxCall(url, function(response){  
    it.next(response);  
  });  
}  
  
var it = main();  
it.next();
```

# | ES6-Generator 应用

- (2) 控制流管理

- 如果有一个多步操作非常耗时，采用回调函数，可能会写成下面这样。

```
step1(function (value1) {  
  step2(value1, function(value2) {  
    step3(value2, function(value3) {  
      step4(value3, function(value4) {  
        // Do something with value4  
      });  
    });  
  });  
});
```

## ES6-Generator 应用

- 采用 Promise 改写上面的代码。

```
Promise.resolve(step1)
  .then(step2)
  .then(step3)
  .then(step4)
  .then(function (value4) {
    // Do something with value4
  }, function (error) {
    // Handle any error from step1 through step4
  })
  .done();
```

## ES6-Generator 应用

- 上面代码已经把回调函数，改成了直线执行的形式，但是加入了大量 Promise 的语法。Generator 函数可以进一步改善代码运行流程。

```
function* longRunningTask(value1) {  
  try {  
    var value2 = yield step1(value1);  
    var value3 = yield step2(value2);  
    var value4 = yield step3(value3);  
    var value5 = yield step4(value4);  
    // Do something with value4  
  } catch (e) {  
    // Handle any error from step1 through step4  
  }  
}
```

## ES6-Generator 应用

- 然后，使用一个函数，按次序自动执行所有步骤。

```
scheduler(longRunningTask(initialValue));  
function scheduler(task) {  
    var taskObj = task.next(task.value);  
    // 如果Generator函数未结束，就继续调用  
    if (!taskObj.done) {  
        task.value = taskObj.value  
        scheduler(task);  
    }  
}
```

- 注意，上面这种做法，只适合同步操作，即所有的task都必须是同步的，不能有异步操作。因为这里的代码一得到返回值，就继续往下执行，没有判断异步操作何时完成。



## | ES6-async

- ES2017 标准引入了 async 函数，使得异步操作变得更加方便。
- async 函数是什么？一句话，它就是 Generator 函数的语法糖。
- async函数就是将 Generator 函数的星号（\*）替换成async，将yield替换成await，仅此而已。

## | ES6-async

- 使用Generator 函数，依次读取两个文件。

```
const fs = require('fs');
```

```
const readFile = function (fileName) {  
  return new Promise(function (resolve, reject) {  
    fs.readFile(fileName, function(error, data) {  
      if (error) return reject(error);  
      resolve(data);  
    });  
  });  
};
```

```
const gen = function* () {  
  const f1 = yield readFile('/etc/fstab');  
  const f2 = yield readFile('/etc/shells');  
  console.log(f1.toString());  
  console.log(f2.toString());  
};
```

## | ES6-async

- 上面的Generator 函数，改写成async函数如下：

```
const asyncReadFile = async function () {  
  const f1 = await readFile('/etc/fstab');  
  const f2 = await readFile('/etc/shells');  
  console.log(f1.toString());  
  console.log(f2.toString());  
};
```

- 一比较就会发现，async函数就是将 Generator 函数的星号（\*）替换成async，将yield替换成await，仅此而已。

# | ES6-async

- async函数对 Generator 函数的改进，体现在以下四点：
- （1）内置执行器。
  - Generator 函数的执行必须靠执行器，所以才有了co模块，而async函数自带执行器。也就是说，async函数的执行，与普通函数一模一样，只要一行。
  - 调用函数它就会自动执行，输出最后结果，不像 Generator 函数，需要调用next方法，或者用co模块，才能真正执行，得到最后结果。
- （2）更好的语义。
  - async和await，比起星号和yield，语义更清楚了。
  - async表示函数里有异步操作。
  - await表示紧跟在后面的表达式需要等待结果。

## | ES6-async

- (3) 更广的适用性。
  - 模块约定, yield命令后面只能是 Thunk 函数或 Promise 对象, 而async函数的await命令后面, 可以是 Promise 对象和原始类型的值 (数值、字符串和布尔值, 但这时会自动转成立即 resolved 的 Promise 对象)。
- (4) 返回值是 Promise。
  - async函数的返回值是 Promise 对象, 这比 Generator 函数的返回值是 Iterator 对象方便多了。你可以用then方法指定下一步的操作。
  - async函数完全可以看作多个异步操作, 包装成的一个 Promise 对象, 而await命令就是内部then命令的语法糖。

# | ES6-async

- 语法

```
async function name(param) { statements }
```

- name: 函数名称。
- param: 要传递给函数的参数的名称。
- statements: 函数体语句。
- async 函数返回一个 Promise 对象，可以使用 then 方法添加回调函数。

```
async function helloAsync() {  
    return "helloAsync";  
}
```

```
console.log(helloAsync()) // Promise {<resolved>: "helloAsync"}
```

```
helloAsync().then(data => {  
    console.log(data);      // helloAsync  
})
```

## | ES6-async

- async 函数中可能会有 await 表达式，async 函数执行时，如果遇到 await 就会先暂停执行，等到触发的异步操作完成后，恢复 async 函数的执行并返回解析值。
- await 操作符用于等待一个 Promise 对象，它只能在异步函数 async function 内部使用。如果在 async function 函数体外使用 await，你只会得到一个语法错误。

## | ES6-async

- async函数返回一个 Promise 对象，可以使用then方法添加回调函数。当函数执行的时候，一旦遇到await就会先返回，等到异步操作完成，再接着执行函数体内后面的语句。
- 示例：指定多少毫秒后输出一个值

```
function timeout(ms) {  
  return new Promise((resolve) => {  
    setTimeout(resolve, ms);  
  });  
}
```

```
async function asyncPrint(value, ms) {  
  await timeout(ms);  
  console.log(value);  
}
```

```
asyncPrint('hello world', 50);
```



## | ES6-async

- 由于async函数返回的是 Promise 对象，可以作为await命令的参数。所以，上面的例子也可以写成下面的形式。

```
async function timeout(ms) {  
  await new Promise((resolve) => {  
    setTimeout(resolve, ms);  
  });  
}
```

```
async function asyncPrint(value, ms) {  
  await timeout(ms);  
  console.log(value);  
}
```

```
asyncPrint('hello world', 50);
```

# | ES6-async

- async 函数有多种使用形式。

- 函数声明

```
async function foo() {}
```

- 函数表达式

```
const foo = async function () {};
```

- 对象的方法

```
let obj = { async foo() {} };
```

```
obj.foo().then(...)
```

# | ES6-async

- Class 的方法

```
class Storage {  
  constructor() {  
    this.cachePromise = caches.open('avatars');  
  }  
  async getAvatar(name) {  
    const cache = await this.cachePromise;  
    return cache.match(`/avatars/${name}.jpg`);  
  }  
}  
  
const storage = new Storage();  
storage.getAvatar('jake').then(...);
```

- 箭头函数

```
const foo = async () => {};
```

## | ES6-async与await

- 正常情况下，await命令后面是一个 Promise 对象，返回该对象的结果。如果不是 Promise 对象，就直接返回对应的值。

```
async function f() {  
    return await 123;    // 等同于 return 123;  
}  
f().then(v => console.log(v))// 123
```

# 华清远见-成都中心-H5教学部



# 目录

客户端JS

支持条件注释的IE

安全性问题

# WEB浏览器-客户端JavaScript

- 呈现静态信息的页面叫做文档；
  - window是web浏览器的一个窗口或窗体，是JavaScript特性和API的主要接入点；
  - window对象是全局对象，处于作用域链的最顶层；
  - 用户的体验不应完全依赖于JS，但JS能显著的提升用户体验；

## WEB浏览器-支持条件注释的IE

- 条件注释在解决IE兼容性问题上非常有用；

- 使用方式：

```
<!--[if lte IE 6]>
```

```
<script src="//misc.360buyimg.com/mtd/pc/index/home/ie6tip.min.js"></script>
```

```
<![endif]-->
```



# WEB浏览器-安全性问题

- JS的解释器在浏览器里，当代码加载完成后，JS就可以做任何事情；
  - 可以任何事情，这样就会存在安全隐患；
  - 浏览器厂商在提供强大的客户端API的时候，也要考虑安全问题；
  - 安全问题类似获取隐私数据、修改删除数据、诈骗、刷流量等；
- 为了安全问题，有些事情JavaScript是不能做的
  - 没有权限操作文件或目录；
  - 不能直接打开新窗口，必须有对应的触发事件，比如鼠标点一下；
  - 可以关闭自己打开的窗口，但是关闭其他窗口必须经过用户同意；

# 第27章 window对象

华清远见-成都中心-H5教学部



# 目录

定时器

定位和导航

浏览器属性Navigator

对话框

## | window对象-定时器

- 定时器是指在指定的时间单次或循环执行一个动作；
  - setTimeout(): 在指定的毫秒数之后执行一个函数；三个参数：  
参数1: 要执行的函数名，不带小括号；  
参数2: 要等待的毫秒数，多少毫秒之后执行；  
参数3: 可忽略，是指给参数1这个函数传递的参数；
  - 只执行一次，只调用一次指定的函数，就一次；
  - 实例：

```
function st (argument) {  
    console.log('a' + argument);  
}  
var stid = setTimeout(st, 2000, '打印我到控制台');
```
  - 如果要想取消这个定时器也是可以的，使用clearTimeout();
  - 实例：

```
clearTimeout(stid);
```

## | window对象-定时器

- setInterval()：在指定的毫秒间隔里重复执行一个函数；
  - 三个参数：  
参数1：要执行的函数名，不带小括号；  
参数2：执行的间隔毫秒数；  
参数3：可忽略，是指给参数1这个函数传递的参数
- 实例：

```
function st (argument) {  
    console.log(new Date().toString() + ': ' + argument);  
}  
//开启一个定时器  
var stid = setInterval(st, 2000, '打印我到控制台');
```
- setInterval() 设置的定时器任务也是可以清除的；  
clearInterval(stid);

# | window对象-定位和导航

- window的location属性引用的是Location对象；
  - href属性是一个字符串，表示当前页面的完整URL地址；
    - 实现页面跳转实例：

```
window.location.href = 'http://www.farsight.com.cn/';
```
  - reload(): 重新加载当前页面；
    - 刷新当前页面实例：

```
window.location.reload();
```
- window.location的更多属性: protocol、host、hostname、pathname、search;

## ！ window对象-浏览器属性navigator

- window的navigator属性返回的是浏览器厂商及版本信息；
  - appName: 浏览器全称；  
IE浏览器是“Microsoft Internet Explorer”；  
其它大多浏览器返回的是“Netscape”，主要是为了代码兼容；
  - appVersion: 包含浏览器厂商和版本信息的字符串；  
字符串开头的数字表示他是第4代或第5代兼容浏览器；  
没有标准格式，不用来判断浏览器；
  - userAgent: 通常包含appVersion里面的所有信息，还有其他相关信息；  
没有标准格式
  - platform: 运行浏览器的操作系统；

## ！ window对象-对话框

- window有三个方法向用户提供简单的对话框；
  - alert():向用户显示一条信息，并等待用户关闭窗口；
  - confirm():显示一个弹窗，要求用户单击确定或者取消，返回true或者false；
  - prompt():显示一个弹窗，等待用户输入字符串，并返回输入的字符串；
- 这三个方法会产生阻塞，也就是在用户做出反应之前，代码不会继续往下执行；





海量视频 贴身学习



超多干货 实时更新

# THANKS

— 谢谢 —